UNIVERSIDADE ESTADUAL DO CEARÁ - UECE
CENTRO DE CIÊNCIAS E TECNOLOGIA – CCT
MESTRADO ACADÊMICO EM CIÊNCIA DA COMPUTAÇÃO

YRLEYJÂNDER SALMITO LOPES

DESENVOLVIMENTO ORIENTADO A MODELOS EM SISTEMAS MULTI-AGENTES COM DIFERENTES ARQUITETURAS INTERNAS DE AGENTE

FORTALEZA - CEARÁ
2012

YRLEYJÂNDER SALMITO LOPES

DESENVOLVIMENTO ORIENTADO A MODELOS EM SISTEMAS MULTI-AGENTES COM DIFERENTES ARQUITETURAS INTERNAS DE AGENTE

Dissertação submetida à Comissão Examinadora do Programa de Pós-Graduação Acadêmica em Ciência da Computação da Universidade Estadual do Ceará, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientação: Professora Dra. Mariela Inés Cortés

FORTALEZA – CEARÁ

2012

YRLEYJÂNDER SALMITO LOPES

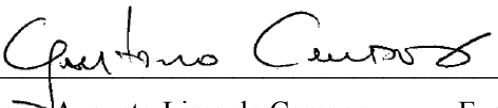
DESENVOLVIMENTO ORIENTADO A MODELOS EM SISTEMAS MULTI-AGENTES COM DIFERENTES ARQUITETURAS INTERNAS DE AGENTE

Dissertação de Mestrado submetida à comissão do Mestrado Acadêmico em Ciência da Computação, da Universidade Estadual do Ceará – UECE, como requisito parcial para obtenção de título de Mestre em Ciência da Computação. Aprovada pela comissão examinadora abaixo assinada.

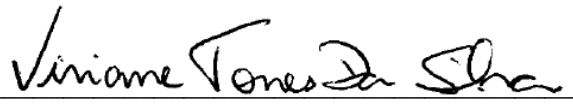
Dissertação aprovada em: ____/____/_____

BANCA EXAMINADORA

Profª. Dra. Mariela Inés Cortés ..... Orientadora
Universidade Estadual do Ceará



Prof. Dr. Gustavo Augusto Lima de Campos ..... Examinador
Universidade Estadual do Ceará



Prof. Dra. Viviane Torres da Silva ..... Examinador
Universidade Federal Fluminense

Dedico este trabalho aos meus pais, José e Aparecida, às minhas irmãs e à minha noiva Estéfanny. Obrigado pela paciência!

AGRADECIMENTOS

Agradeço pela oportunidade de desenvolver e crescer, os sinceros agradecimentos a todas as pessoas que de alguma maneira colaboraram para a realização deste trabalho, em especial:

A Deus, por estar sempre presente em minha vida, guiando os meus passos, suprindo necessidades e me dado este presente.

Aos meus pais e irmãs, por acreditarem em mim, pelo afeto, apoio e por sempre me ajudarem a realizar meus sonhos.

A minha noiva pela paciência de suportar a distância muitas vezes em prol do nosso sucesso.

Aos meus amigos, colegas e professores do Mestrado em Ciência da Computação, pelos trabalhos que fizemos juntos e lições aprendidas.

À professora Mariela Cortés, orientadora desta dissertação, pelo seu apoio, empenho, paciência, descontração e preparo profissional, fundamentais para a conclusão desta.

Ao meu amigo e grande colaborador deste trabalho Enyo Gonçalves, pela incontável disponibilidade das reuniões que tivemos.

Ao meu colega de mestrado Sávio Freire, pelo apoio a distância em resolvermos assuntos referentes às nossas pesquisas e trabalhos.

Enfim, a todos que de alguma forma contribuíram para que este trabalho fosse concretizado através dos incentivos recebidos.

"O planejamento não é uma tentativa de predizer o que vai acontecer. O planejamento é um instrumento para raciocinar agora, sobre que trabalhos e ações serão necessários hoje, para merecermos um futuro. O produto final do planejamento não é a informação: é sempre o trabalho. Sessenta por cento de todos os problemas administrativos resultam da ineficácia da comunicação."

PETER DRUCKER.

RESUMO

A Engenharia de *Software* Orientada a Agentes apresenta métodos, técnicas e ferramentas que possibilitam o desenvolvimento de sistemas centrados em agentes de forma satisfatória. No contexto do desenvolvimento de software, a fase de implementação tem por objetivo a codificação do sistema em consistência com o projeto detalhado, documentado ao longo de um conjunto de modelos. Neste cenário, a consonância entre as ferramentas envolvidas na geração dos artefatos de modelagem e de código é fundamental para reduzir o *gap* semântico entre estes dois níveis de abstração. Como auxílio, o desenvolvimento dirigido por modelos pode colaborar em evitar o retrabalho na geração dos artefatos necessários para Sistemas Multi-Agentes (SMA).

O presente trabalho de dissertação propõe a evolução do framework JADE, que possui um ambiente de execução dos agentes e que pode ser aplicado à introdução de conceitos de SMA em relação às arquiteturas internas. A ferramenta MAS-ML *Tool* também está envolvida, de forma a possibilitar a geração de código de SMA em JADE a partir de modelos de MAS-ML 2.0, pois esta linguagem de modelagem adequa a interação entre agentes e ambientes de forma satisfatória. O benefício alcançado com esta abordagem é o aumento de produtividade e padronização do código gerado, além de possibilitar a consistência entre os modelos e código e a rastreabilidade entre os artefatos.

Palavras-Chave: Sistema Multi-Agente, Desenvolvimento orientado a modelos, MAS-ML, JADE.

ABSTRACT

Agent-based Software Development presents methods, techniques and tools that enable the development of systems to agents satisfactorily. In the context of software development, the implementation phase aims coding system consistent with the detailed design, documented over a set of models. In this scenario, the synchronization between the tools involved in the generation of modeling artifacts and code is essential to reduce the semantic gap between these two levels of abstraction and helping, the model driven development can collaborate to reduce the rework in the generation of required artifacts for Multi-Agent Systems (MAS).

This paper proposes the evolution of the JADE framework, that has an environment for execution of agents and can be applied to introduce the MAS concepts related to the internal architectures. The MAS-ML Tool is also involved in order to enable the generation of SMA code in JADE from MAS-ML 2.0 models, because this modeling language appropriates the interaction between agents and environment in a satisfactory way. The benefit achieved with this approach is the increased productivity and standardization of the generated code, and providing consistency between models and code, and traceability between artifacts.

Keywords: *Multi-agent system, Model driven architecture, MAS-ML, JADE*.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

ACL	*Agent Comunication Language*
AMS	*Agent Management System*
AORML	*Agent Object Relationship Modeling Language*
AOS	*Agent Oriented Software*
AUML	*Agent Unified Modeling Language*
AVA	Ambiente Virtual de Aprendizagem
BDI	*Belief-Desire-Intention*
CASE	*Computer Aided Software Engineering*
CIM	*Computation Independent Model*
CMS	*Course Management System*
CWM	*Common Warehouse Metamodel*
DF	*Directory Facilitator*
EMF	*Eclipse Modeling Framework*
FIPA	*Foundations of Intelligent Physical Agents*
GEF	*Graphical Editing Framework*
GMF	*Graphical Modeling Framework*
IDE	*Integrated Development Environment*
ISM	*Implementation Specific Model*
JACK	*JACK Agent Language*
JAMDER	*JADE to MAS-ML 2.0 Development Resource*
LGPL	*Lesses General Public License Version 2*
LMS	*Learning Management System*
MAS-ML	*Multi Agent System – Modeling Language*
MDA	*Model Driven Architecture ou Arquitetura Orientada por Modelos*
MOF	*Meta Object Facility*
MTL	*Model Transformation Language*
OMG	*Object Management Group*

PDM	*Platform Definition Model*
PDT	*Prometheus Design Tool*
PIM	*Platform Independent Model*
PRS	*Procedural Reasoning System*
PSM	*Platform Specific Model*
SMA	Sistemas Multi-Agentes
SDK	*Software Development Kit*
TAO	*Taming Agents and Objects*
TI	Tecnologia da Informação
UML	*Unified Modeling Language*
VTL	*Velocity Template Language*
XMI	*XML Metadata Interchange*
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Desenvolvimento de software orientado a agentes é uma abordagem que vem conquistando espaço tanto na indústria quanto no meio acadêmico, visto que inteligência, autonomia e dinamismo são altamente desejáveis e as aplicações estão ficando cada vez mais complexas. Em SMA (Sistemas Multi-Agentes), o agente é a entidade central capaz de perceber seu ambiente por meio de sensores, agir neste ambiente por intermédio de atuadores [RUSSELL; NORVIG, 2004] e, dependendo de sua arquitetura interna, adquirir conhecimento, guardar o histórico de ações, entre outras propriedades. A expressão Sistema Multi-Agentes refere-se à subárea de Inteligência Artificial que investiga o comportamento de um conjunto de agentes autônomos objetivando a solução de um problema que está além da capacidade de um único agente [JENNINGS, 1999]. Além de perceber e agir, os agentes de um SMA são capazes de comunicar-se com outros agentes e decidir sua atuação.

A arquitetura interna do agente define suas propriedades (componentes mentais e comportamentais), determinando, consequentemente, uma implementação diferenciada para cada caso. As arquiteturas internas de agente definidas por Russell e Norvig [2004] são as seguintes: agente reativo simples, agente reativo baseado em conhecimento, agente baseado em objetivo e agente baseado em utilidade. Por exemplo, a atribuição de crenças (estados do mundo) e a utilização de uma função-próximo para auxiliar na escolha da próxima ação a ser executada são propriedades inerentes ao agente reativo baseado em conhecimento [WEISS apud GONÇALVES, 1999]. Os agentes dotados de uma destas arquiteturas internas são conhecidos como agentes inteligentes.

Um SMA não é composto apenas por agentes [SILVA, 2004]. Outras entidades podem ser utilizadas e relaciondas, como papel de agente, organização, ambiente, papel de objeto e objeto, cada uma com sua importância para o funcionamento do SMA.

A utilização de ferramentas CASE (*Computer Aided Software Engineering*) [SOMMERVILLE, 2007] em SMA dá apoio ao processo de desenvolvimento, automatizando as atividades de forma a facilitar e agilizar o trabalho dos profissionais envolvidos. Mais especificamente, ferramentas de modelagem sustentam as atividades de análise e projeto, enquanto que *frameworks* e

plataformas propiciam o ambiente para a implementação. Neste contexto, é desejável que exista uma consonância entre estas duas fases no intuito de diminuir o *gap* semântico existente entre elas, de forma a facilitar a geração de código a partir dos modelos gerados na fase de projeto.

Desta forma se torna necessário estabelecer um mapeamento entre análise, projeto e implementação, levando em consideração que os artefatos gerados como resultado possuem níveis diferentes de abstração. A definição de um mapeamento estabelece como as entidades de um SMA propostas em nível de modelagem podem ser implementadas no contexto de um framework ou linguagem de implementação. A partir da definição do mapeamento, é interessante que o processo de geração de código seja automatizado, tendo como base uma ferramenta, de forma a diminuir o tempo de desenvolvimento, e, adicionalmente, fornecer mecanismos de checagem que garantam a geração das entidades corretamente e com maior agilidade.

1.1 Problema

Abordagens para a geração de código automaticamente a partir da modelagem são escassas. Por outro lado, embora vários *frameworks* orientados a agentes tenham sido propostos, nenhum deles aborda adequadamente os diferentes tipos de arquiteturas internas de agentes inteligentes[1], ou, ainda não se encontram em consonância com as demais entidades que tipicamente são contempladas nas linguagens de modelagem (como organização e papel de agente, por exemplo).

A existência de um *framework* que contemple as diversas arquiteturas se faz necessária dada a necessidade de desenvolver SMAs onde agentes com diferentes características e propriedades possam interagir de forma sincronizada e coerente. A consonância entre as entidades presentes na modelagem e no código permite reduzir o *gap* existente entre as fases de análise e projeto e a fase de desenvolvimento. Além disto, a geração de código apartir de modelos é importante, pois reduz o tempo de codificação e propicia a consistência entre os modelos e o código gerado.

[1] Um agente inteligente (ou agente racional) é aquele que executa a ação correta [RUSSELL e NORVIG, 2004].

1.2 Objetivos

O adequado mapeamento das representações de entidades e seus relacionamentos em SMA, envolvendo modelagem e implementação, torna-se necessário para que o desenvolvimento de sistemas direcionados a agentes seja elaborado de forma mais adequada e concisa. Isto inclui o fato de que, um projeto SMA pode incluir agentes com diferentes tipos de arquitetura, tornando o projeto mais complexo e propenso a erros, e consequentemente, o tempo de desenvolvimento mais prolongado.

Este trabalho tem como objetivo geral, promover a geração de código a partir de artefato de modelagem que contemple todas as entidades tipicamente encontradas em SMA, assim como as diversas arquiteturas de agente. A abordagem propõe a geração a partir de uma linguagem de modelagem para um *framework* de implementação, ambos orientados a agentes, levando em conta as entidades de um SMA e características das entidades contempladas e em particular das arquiteturas internas de agente, modeladas na linguagem escolhida.

O objetivo do trabalho envolve o estabelecimento dos principais elementos considerados em nível de linguagem de modelagem a serem contemplados na geração dos artefatos de código. Este trabalho também contempla o estudo dos demais elementos que normalmente fazem parte de um SMA, mas particularmente, adaptados por Silva [2004] e Gonçalves [2009], a saber: o ambiente, organização, papel de agente, papel de objeto e objeto.

A abordagem para o desenvolvimento de software orientado a modelos prevê a automatização para a fase de implementação contemplando todas as entidades previstas na fase de modelagem, através de um gerador de código que irá concretizar esta transformação de modo a contemplar as arquiteturas internas dos agentes inteligentes. Com isso, os profissionais envolvidos no desenvolvimento de SMAs poderão se beneficiar da adequação entre a modelagem de sistemas e sua implementação reduzindo o tempo gasto e a propensão a erros típicos na geração de código realizada manualmente.

1.3 Justificativa

O processo de analisar um problema, encontrar uma solução, e expressá-la em uma linguagem de programação de alto nível, pode ser visto como uma forma implícita de modelagem [FRANCE; RUMPE, 2007]. A partir desta premissa, no intuito de desenvolver sistemas baseados em agentes, é necessário verificar se a modelagem e implementação escolhidas para o desenvolvimento contemplam as mesmas características.

O conceito primordial do padrão orientado a agentes é o próprio agente. Dependendo da arquitetura interna [RUSSELL; NORVIG, 2004] do agente ou mesmo do seu papel a ser executado, sua modelagem e consequentemente sua implementação, devem ser adequadas. Por isto, é importante a adequação entre ferramentas de modelagem e implementação envolvidas, caso não exista.

A possibilidade de geração de código vinculado às ferramentas de modelagem no domínio de SMA é rara, porém, a geração automática de código no desenvolvimento de software traz vários benefícios, entre eles:

- Refletir no código as propriedades estruturais e comportamentais das entidades de acordo com a sua modelagem;
- Boas práticas de programação: códigos padronizados são gerados (semi) automaticamente para uma determinada situação. Isto evita problemas relacionados à omissão de alguma característica estrutural ou comportamental no código;
- Redução do custo de manutenibilidade: alterações dos artefatos de modelagem podem ser incorporadas diretamente nos artefatos da codificação a partir da geração de código;
- Rastreabilidade entre os artefatos de modelagem e código;

1.4 Metodologia

O embasamento teórico deste trabalho é fundamentado em livros, artigos de periódicos, dissertações de mestrado e teses de doutorado relacionadas ao tema em questão e informações obtidas com pesquisadores de instituições de ensino com competência na área.

Com base no levantamento teórico realizado, foi elaborada uma teoria formalmente fundamentada com o intuito de verificar os elementos relacionados à modelagem de arquiteturas internas de agente e às classes do *framework* de implementação.

De acordo com o estudo preliminar dos trabalhos relacionados há uma tendência natural em relação às ferramentas envolvidas nas diversas etapas de desenvolvimento de software e do padrão de geração de código. É possível estabelecer um conjunto de ferramentas candidatas a serem utilizadas nas respectivas etapas, por serem mais adequadas ao propósito deste trabalho. Dentre as ferramentas MDA para geração de código foi escolhido o *plugin* Acceleo, o qual tem bastante material de apoio para dar suporte à transformação dos modelos gerados a partir da ferramenta MAS-ML *Tool* em código.

A validação da proposta envolve a realização de estudo de caso que ilustra o processo de geração de código a partir dos artefatos de modelagem MAS-ML 2.0.

Alguns artigos foram escritos a partir dos resultados obtidos, gerando algumas publicações [GONÇALVES et al., 2010a], [LOPES et al., 2011a] e [LOPES et al., 2011b], e esta dissertação é o resultado final deste trabalho.

1.5 Estrutura da Dissertação

Esta dissertação está organizada da seguinte maneira:

O Capítulo 2 apresenta os trabalhos relacionados aos *frameworks* de implementação em SMA e a geração de código para SMA;

O Capítulo 3 apresenta o referencial teórico da linguagem de modelagem MAS-ML 2.0, do *framework* de implemetação escolhido, JADE, e, uma introdução ao conceito de MDA (*Model Driven Architecture*) ;

O Capítulo 4 descreve a extensão proposta ao JADE, o *framework* JAMDER;

O Capítulo 5 aborda a geração de código a partir das ferramentas utilizadas e dos *templates* de geração de código em Acceleo;

O Capítulo 6 apresenta o estudo de caso ilustrando o processo de geração de código segundo a abordagem aqui proposta para um sistema baseado em SMA;

Finalmente, o Capítulo 7 contém as considerações finais e os trabalhos futuros.

2 TRABALHOS RELACIONADOS

A análise dos trabalhos relacionados a esta proposta envolve um levantamento e comparativo entre aos principais *frameworks* de implementação orientados à agente e ferramentas utilizadas para geração de código. Várias linguagens de modelagem e *frameworks* de implementação têm sido propostos para desenvolver SMAs, porém, a escolha pelos mecanismos a serem adotados em cada etapa do presente trabalho depende da sua adequação ao domínio do problema e aos objetivos aqui propostos.

Assim como no desenvolvimento orientado a objetos, o paradigma de desenvolvimento orientado a agentes requer técnicas adequadas que explorem seus benefícios e suas características próprias, dando suporte na produção e manutenção deste tipo de *software* [ZAMBONELLI; JENNINGS; WOOLDRIDGE, 2001]. A seguir são apresentadas algumas ferramentas de implementação em SMA.

2.1 Frameworks de Implementação

Um conjunto de *frameworks* e plataformas tem sido desenvolvido no intuito de dar apoio ao desenvolvimento de SMAs. De forma geral, estes mecanismos estão associados a uma linguagem de programação para compor as entidades e fornecem um ambiente para a execução dos mesmos. A seguir são apresentados alguns dos mais utilizados *frameworks* de implementação para SMA.

2.1.1 JACK

JACK [JACK, 2011] é um *framework* em Java para o desenvolvimento de sistemas multi-agentes e construído pela AOS (*Agent Oriented Software*). Esse *framework* oferece alto desempenho, e uma maneira fácil de ser estendido para dar suporte a agentes BDI (*Belief-Desire-Intention*) e requisitos específicos das aplicações. A linguagem utilizada pelo JACK (*JACK Agent Language*) é construída a partir da linguagem Java podendo ser usada no desenvolvimento de agentes [NUNES, 2007].

A linguagem JACK permite que os programadores desenvolvam componentes que são necessários aos agentes BDI [RAO; GEORGEFF, 1995] e o seu comportamento. As unidades funcionais presentes são:

- *Agent* – definir o comportamento de um agente de software inteligente.
- *Capability* – Capacidades representam aspectos funcionais de um agente que podem ser plugadas quando necessárias.
- *BeliefSet* – Representa as crenças do agente, fazendo uso de um modelo relacional genérico.
- *View* – Permite que consultas de propósito geral possam ser feitas sobre o modelo de dados.
- *Event* – Descreve uma ocorrência, para a qual o agente deve tomar uma ação como resposta.
- *Plan* – São as instruções que o agente segue para tentar atingir seus objetivos e tratar os eventos designados a ele.

Os agentes JACK são componentes de software autônomos que têm objetivos explícitos para atingir ou eventos para tratar. Jack oferece suporte a agentes baseados em BDI.

Alguns pontos negativos de JACK:

- A ferramenta IDE (*Integrated Development Environment*) de JACK não é gratuita por possuir uma forte preocupação com aplicações industriais;
- JACK define que os agentes têm um conjunto de crenças e objetivos [NUNES, 2007], portanto esta ferramenta não dá suporte a agentes reativos simples;
- Não foi encontrado algo em relação a papel de agente ou organização.

2.1.2 JAM

Este framework para agentes inteligentes foi desenvolvida com base em uma série de teorias sobre agentes e *frameworks* para agentes baseadas em BDI. Entre essas teorias temos: as arquiteturas de agentes inteligentes do PRS (*Procedural Reasoning System*) [HUBER, 2011] e sua implementação chamada UMPRS. A

arquitetura JAM [HUBER, 2011] é composta de cinco componentes primários [NUNES, 2007]:

- *World Model* – banco de dados que representa as crenças do agente.
- *Plan Library* – coleção dos planos que o agente pode usar para atingir seus objetivos.
- *Interpreter* – considerado o cérebro dos agentes. É quem raciocina.
- *Intention Structure* – modelo interno dos objetivos e das atividades correntes do agente.
- *Observer* – considerado um plano leve que é executado entre os passos do plano, com o propósito de realizar funcionalidades que não estão determinadas no curso normal do plano.

A arquitetura JAM é implementada em Java. Existem funcionalidades que ainda não foram implementadas nessa arquitetura, tais como geração de planos e aprendizado.

Alguns pontos negativos de JAM:

- Não possui ferramenta de desenvolvimento (IDE);
- Como JAM define em sua estrutura o componente *Intention Structure*, que é uma representação de objetivo, então ela não suporta o conceito de agente reativo simples. E também foi encontrado nenhuma referência a papel ou organização;

2.1.3 JADE

JADE [JADE, 2011] é um *framework* totalmente implementado na linguagem Java, simplificando o desenvolvimento de sistemas multi-agentes através do *middle-ware* que está de acordo com as especificações FIPA (*Foundations of Intelligent Physical Agents*) [FIPA, 2011] e com um conjunto de ferramentas gráficas que suportam *debugging* e frases de organização [NUNES, 2007].

A plataforma de agentes pode ser distribuída através de máquinas e a configuração pode ser controlada por uma interface gráfica remota.

A configuração pode até mesmo ser mudada em tempo de execução movendo agentes de uma máquina a outra, como e quando necessário. O único requisito para utilizar o *framework* JADE é o Java SDK (*Software Development Kit*) 1.4 ou superior.

JADE é *software* livre e é distribuído pela Telecom Itália, em um código fonte aberto sobre os termos da licença LGPL (*Lesses General Public License Version 2*).

JADE é amplamente utilizado no mercado em vários setores como: telecomunicação (*Telecom Italia LAB*), aplicações de internet (*Caboodle Networks*), etc. Sua eficácia aumenta a cada versão atualizada e possui bom *feedback* das empresas que o utiliza. É *open source*, possuindo a linguagem Java como implementação, o que facilita ser usado em qualquer sistema operacional dotado de JVM (*Java Virtual Machine*).

Alguns pontos negativos de JADE:

- Não dá suporte a agentes proativos (baseados em objetivos e utilidade) por não possuir o componente objetivo em sua estrutura, assim como os métodos necessários para este tipo de agente;
- Não foi encontrado referência em relação a papéis de agente;

2.1.4 JADEX

JADEX [JADEX, 2011] é uma plataforma de desenvolvimento orientado a agentes, no qual agentes racionais são escritos em XML (*Extensible Markup Language*) e na linguagem de programação Java. O JADEX pode ser programado nos ambientes de desenvolvimento integrado orientados a objetos [NUNES, 2007].

O JADEX é independente, e pode ser usado com diferentes plataformas de agentes. O primeiro é a plataforma JADE, a qual é bem conhecida e de código livre. O segundo é o adaptador *JADEX Standalone*, o qual é um ambiente pequeno, mas rápido.

JADEX segue o modelo BDI. No JADEX, agentes têm crenças, que são armazenadas em uma base de crenças, objetivos representam motivações concretas, e para atingir seus objetivos, o agente executa planos, os quais são roteiros procedurais codificados em Java.

Os principais componentes do JADEX são:

- *Capability* – Capacidades permitem que crenças, planos e objetivos sejam colocados em um módulo de agente.
- *Beliefs* – Crenças representam o conhecimento do agente sobre o seu ambiente e sobre si mesmo. Em JADEX, as crenças podem ser qualquer objeto Java.
- *Goals* – Objetivos compõem a postura motivacional do agente, são o que orientam suas ações. Objetivo é um conceito central no JADEX. Objetivos são desejos concretos e momentâneos de um agente.
- *Plans* – Planos representam a forma como o agente atuará em seu ambiente. Planos são selecionados em resposta à ocorrência de eventos ou de objetivos.
- *Events* – Agentes possuem a capacidade de reagir a diferentes tipos de eventos. JADEX suporta dois tipos de eventos: eventos internos que podem ser usados para denotar uma ocorrência dentro de um agente, eventos mensagem que representam uma comunicação entre dois agentes ou mais. Eventos normalmente são tratados por planos.

O JADEX consiste de dois componentes entrelaçados. Primeiro, o agente reage às mensagens que chegam, a eventos internos e a objetivos. Segundo, o agente continuamente delibera seus objetivos correntes, para decidir a respeito de um subconjunto consistente, o qual deve ser perseguido.

O JADEX não introduz uma nova linguagem de programação de agentes, ele usa técnicas de engenharia de software estabelecidas, como Java e XML. JADEX possui uma série de ferramentas disponíveis. Que permite o controle da execução dos agentes, verificando os dados, tais como suas crenças, objetivos e planos, depuração e documentação.

Alguns pontos negativos em relação ao JADEX:

- Plataforma ainda em fase de maturidade, ou seja, cada atualização apresenta uma série de modificações não compatíveis com a versão anterior;

- O uso de XML para criar características de agentes, aumenta a complexidade do desenvolvimento, por ter outra estrutura de programação;

2.1.5 Jason

Jason [JASON, 2011] é uma plataforma de desenvolvimento de sistemas multi-agentes baseada em um interpretador para uma versão estendida da linguagem AgentSpeak(L), que é uma linguagem de programação orientada a agentes baseada na lógica de primeira ordem, com eventos e ações. Além disso, ela é baseada em implementações já existentes de sistemas BDI [NUNES, 2007].

A linguagem AgentSpeak(L) tem como especificações:

- *Beliefs Base* – conjunto de crenças base, as quais são fórmulas atômicas de primeira ordem.
- *Goal* – é um estado do sistema em que o agente deseja chegar. Existem dois tipos de objetivos:
  - *Achievement Goal*: o agente deseja atingir o estado de mundo onde a fórmula atômica associada é verdadeira.
  - *Test Goal*: o agente deseja testar se a fórmula atômica associada é uma de suas crenças.
- *Plan Library* – é uma biblioteca de planos do agente, na qual cada plano determina um conjunto de ações que deve ser executado a fim de um dado objetivo ser atingido. Cada plano é constituído de:
  - *Head*: formado por um evento ativador (*triggering event*), o qual define quais eventos podem iniciar a execução de um plano e um conjunto de literais representando um contexto.
  - *Body*: O corpo do plano inclui ações básicas, ou seja, ações que representam operações atômicas que o agente pode executar a fim de alterar o ambiente.

Jason é implementado em Java e está disponível em código aberto, podendo-se optar pela execução em um ambiente distribuído. Jason possui um ambiente de desenvolvimento integrado (IDE) que permite a execução de programas também em

modo de depuração. Uma vantagem do uso da linguagem AgentSpeak é que ela possui semântica formal, o que possibilita a verificação formal de sistemas programados com esta linguagem.

Alguns pontos negativos em relação ao Jason:

- Não oferece o recurso de troca de papéis entre agentes;
- Direcionado para agentes BDI;

2.1.6 Comparativo dos *frameworks* de implementação e linguagens de modelagem de SMAs

De todos os *frameworks* propostos, apenas JADE, Jason e JADEX possuem uma IDE gratuita com suporte a depuração de código. Como em qualquer um deles seria necessário a extensão para melhor se adequar à linguagem de modelagem, então JADE mostrou a mais adequada por não precisar redefinir estruturas de XML como no JADEX [JADEX, 2011] e por não ser direcionada para agentes BDI, como JADEX e Jason, possibilitando sua extensão para este tipo de agente.

Resumindo, as ferramentas de implementação, de acordo com a Tabela 1, temos os seguintes critérios comparativos:

Tabela 1 Tabela Comparativa dos *frameworks* de implementação

Critérios	**Ferramentas**				
	JACK	**JAM**	**JADE**	**JADEX**	**Jason**
Linguagem	Java	Java	Java	Java e XML	Java
Ferramenta de Desenvolvimento	IDE não é gratuita	-	IDE,execução e *Debug* e documentação	IDE, *debug* e documentação	IDE
Aplicações Comerciais	Tráfego Aéreo e *Real-time Scheduling*	-	Aplicações: móveis, Internet, corporativas e *Machine-to-Machine*	MedPAge, Dynatech e *Bookstore*	-
Outras Características	Não é livre, preocupação com indústria	-	Software Livre, FIPA-*Compliant*, Serviço de páginas amarelas, suporte a ontologias	FIPA-*Compliant*, facilidade de integração com ontologias	Software Livre, Semântica Formal
Arquiteturas suportadas	BDI	BDI	Agentes reativos	BDI	BDI

Os seguintes motivos levaram a esta escolha do framework de desenvolvimento JADE:

- Possuir uma plataforma e linguagem Java que facilita o aprendizado;
- Dá suporte a sistemas distribuídos;
- Está de acordo com os padrões da FIPA;
- Possui interface gráfica e *plugins* para IDEs;
- Gratuito;

A análise das linguagens de modelagem encontra-se no Apêndice I desta dissertação. A escolha da linguagem de modelagem MAS-ML 2.0 ocorreu por esta possuir as características necessárias para a modelagem de SMAs e oferecer suporte as arquiteturas internas de agente.

2.2 Gerador de código existente para SMA

Em relação ao aspecto de geração de código para SMA, os seguintes trabalhos podem ser menecionados.

2.2.1 SMA Modeler

O trabalho de Santos [2008] fornece um metamodelo que busca possibilitar a representação dos conceitos que compõem um agente de *software*, assim como os relacionamentos entre eles. De acordo com Santos [2008], o metamodelo possui:

- A independência com diferentes abordagens de desenvolvimento e plataformas de implementação;
- Possibilita criar modelos de SMA e mapear estes modelos para os conceitos e relacionamentos do metamodelo propostos pelo autor;
- Traduz os modelos de maneira automática para código fonte no *framework* de implementação, *Semanticore* [BLOIS; LUCENA, 2004].

A geração de código proposta por Santos [2008] utiliza um protótipo elaborado na ferramenta Velocity [VELOCITY, 2012], em que para cada *framework* de implementação é necessário:

- Mapeamento dos conceitos e relacionamentos do metamodelo para a plataforma de implementação escolhida.

- Criação de arquivos VTL (*Velocity Template Language*) usados na geração de código para determinada plataforma de implementação.
- Conforme a plataforma de implementação usada, pode ser necessária a criação de classes que auxiliem na geração de código.

Em relação à modelagem, ela é representada através de um arquivo XML que contém a estrutura dos agentes. O autor utilizou uma ferramenta desenvolvida pelo próprio autor, denominada *SMA Modeler*, porém referenciada em seu trabalho como protótipo, onde cada inclusão das informações é posta através de telas passo-a-passo até incluir as informações necessárias. O usuário tem a possibilidade de utilizar a opção *CheckModel* que verifica se a estrutura está consistente. Ao final, esta estrutura fica armazenada em um arquivo XML e que pode ser utilizada pelo *template* de *Velocity* para geração de código. Vale ressaltar que a estrutura deve estar baseada no metamodelo definido pelo autor.

Todavia, o metamodelo definido apenas trata das arquiteturas internas definidas dos agentes em [WOOLDRIDGE, 2002], não contempla o SMA por completo, ou seja, nada é gerado em relação às outras entidades em SMA, tais como ambiente, papéis e organização. A ausência de uma ferramenta de modelagem que possibilite o *design* do modelo do SMA também foi considerada como uma desvantagem. Não foi mencionado algo sobre a arquitetura MDA como forma abordagem de desenvolvimento baseado em modelagem.

Outra ferramenta de modelagem que possibilita a geração código é o ANote [CHOREN; LUCENA, 2004]. O código gerado utiliza o *framework* orientado a objetos ASYNC [CHOREN; LUCENA, 2004], porém o tipo de agente encontrado neste *framework* é apenas do tipo baseado em objetivos guiado por planos. Outro ponto em que ANote não é utilizado neste trabalho se deve ao fato de que não define ambientes.

2.2.2 Prometheus Design Tool (PDT)

Esta ferramenta foi proposta para dar suporte à metodologia Prometheus [PADGHAM, THANGARAJAH and WINIKOFF, 2008]. Segundo Farias et al. [2009], esta ferramenta foi derivada a partir da definição de MAS-ML e como recursos, esta ferramenta apresenta:

- Interface gráfica juntamente com descritor de texto estruturado, onde são informadas as propriedades das entidades;
- Permite a propagação de informações de uma fase do desenvolvimento para outra. Por exemplo, se um objetivo é associado ao papel e este papel a um agente, então objetivo está associado ao agente.
- Visão hierárquica do processo de definição dos agentes, ou seja, é possível ter a definição dos agentes em diferentes níveis de abstração.

Uma vantagem do PDT é que ele pode ser integrado com o Eclipse [ECLIPSE, 2011] através do plugin PDT. Isto permite a integração com outras ferramentas que venham a ser utilizadas também nesta plataforma de desenvolvimento.

A linguagem de implementação utilizada na geração de código por esta ferramenta é a JACK. Porém JACK apenas contempla agente BDI.

2.2.3 Visual Agent

De Maria [2004] propõe a geração de código baseado na arquitetura MDA para sistemas SMA utilizando a linguagem de modelagem MAS-ML e o framework de implementação ASF nas etapas desta arquitetura. O ASF contempla as entidades tipicamente encontradas em SMAs além dos agentes, como: organização, ambiente, papel de agente, papel de objeto e objeto. Cada entidade é representada por um conjunto de classes, ou módulos. A ferramenta Visual Agent é utilizada para a modelagem e geração de código.

Uma grande vantagem é que os conceitos adotados na modelagem MAS-ML são encontrados na linguagem de implementação prosposta.

Utilizando as mesmas linguagens e no context de MDA, o trabalho de Fazziki, Nouzri e Sadgal [2010] utiliza estas linguagens para enfatizar as etapas sugeridas na arquitetura MDA, a fim de promover a independência das linguagens, modelagem e implementação, adotadas no desenvolvimento de SMAs. Porém, neste trabalho não foi adotada uma ferramenta de execução do SMA, utilizando a abordagem.

O trabalho de De Maria [2004] e Fazziki, Nouzri e Sadgal [2010] estão bem próximos à proposta que esta dissertação deseja alcançar em relação à geração de código. Porém, alguns pontos foram considerados:

- O agente analisado tanto na modelagem quanto na implementação apenas considera o agente baseado em objetivo com plano, ou seja, não considerada às diferentes arquiteturas internas de agentes da literatura;
- Para ocorrer a geração de código em ASF a partir do artefato de modelagem MAS-ML, há uma segunda transformação deste artefato, onde ele é transformado em um arquivo do formato de UML (XMI), e então, poder ser transformado em código Java (ASF). A figura mostra este processo.
- Caso o desenvolvedor necessite utilizar outro framework de implementação para MAS-ML, é necessário adaptar os conceitos entre estas duas ferramentas, porém não é possível utilizar a ferramenta Visual Agent, visto que ela apenas possui mecanismos para gerar código em ASF.

3 REFERENCIAL TEÓRICO

A linguagem MAS-ML 2.0 [GONÇALVES, 2009] e o *framework* JADE [JADE, 2011] provêem os mecanismos necessários para apoiar o desenvolvimento de SMAs. Essas ferramentas fornecem suporte em duas fases diferentes do processo de desenvolvimento. A linguagem MAS-ML 2.0, baseada no *framework* conceitual TAO (*Taming Agents and Objects*) [SILVA, 2004] tem seu foco nas fases de análise e projeto, enquanto que o *framework* JADE, na fase de codificação.

MAS-ML 2.0 suporta diretamente os conceitos de agente, organização, papel de objeto, ambiente, papel de objeto e objeto. Diferentemente, o *framework* JADE dá suporte a agente e comportamento de agente (*behaviour*), mas possui uma plataforma de execução e conteineres onde os agentes são executados. Este *framework* provê um arcabouço para o desenvolvimento de sistemas multi-agente de acordo com o padrão FIPA [FIPA, 2011].

O *framework* JADE possui classes e serviços que facilitam a fase de codificação de um SMA. Dentre as funcionalidades disponíveis, algumas são listadas a seguir: publicação de serviços por meio de páginas-amarelas, serviço de localização por meio de páginas-brancas, suporte a ontologias e protocolos de comunicação compatíveis com o padrão FIPA. Considerando que JADE é um *framework* orientado a agentes baseado em uma linguagem orientada a objetos (Java), o conceito das diferentes arquiteturas internas dos agentes de MAS-ML 2.0 deve ser mapeado para objetos providos por JADE.

A seguir estão as definições das arquiteturas internas definidas por Russell e Norvig [2004] e das linguagens e ferramentas abordadas para o desenvolvimento de um SMA.

3.1 Arquiteturas internas de agentes

Russell e Norvig [2004] definiram que o agente atua no ambiente através de suas ações e percebe o ambiente através de seus sensores. Internamente, a função de agente para um agente artificial é implementada por um programa de agente. Dependendo do ambiente de tarefa, a percepção pode não ser fácil para o agente,

onde é preciso que as propriedades do ambiente interno sejam adequadas às propriedades do ambiente externo para que os objetivos sejam realizados.

As arquiteturas internas do agente são basicamente determinadas a partir de dois princípios: reativo e cognitivo. O reativo apenas responde às mudanças que ocorrem em seu ambiente [WEISS, 1999]. Os cognitivos ou pró-ativos podem tomar decisão sobre suas ações, definir sequência de ações e adquirir conhecimento através das situações em que se encontra.

A seguir, mais detalhes sobre as arquiteturas internas definidas por Russell e Norvig [2004].

3.1.1 Agentes Reativos Simples

Este tipo de agente seleciona a ação com base na atual percepção, não registra o histórico de ações anteriores, ou seja, respondem continuamente à mudança de seu ambiente.

Mais especificamente, o funcionamento da arquitetura do agente padrão envolve quatro passos, ou seja: (1) por meio dos sensores, o agente recebe informações do ambiente que são sequências de estados definidos em **S** = $\{s_1, ..., s_n\}$; (2) um subsistema de percepção, **Ver: S** $\rightarrow$ **P**, processa cada estado de uma sequência **S*** e mapeia em uma ação de m percepções, ***P*** = $\{p_1, ..., p_m\}$, que são representações de aspectos dos estados de S que estão acessíveis ao agente para a tomada de decisão; (3) um subsistema de tomada de decisão, Ação: P* $\rightarrow$ A, processa as sequências perceptivas P*, resultantes de S*, e seleciona uma de **I** ações em ***A*** = $\{a_1, ..., a_I\}$; (4) por meio de atuadores o agente envia a ação selecionada para o ambiente [GONÇALVES, 2009].

Russel e Norvig [2004] incorporaram um conjunto de regras condição-ação no subsistema de tomada de decisão do agente reativo. Estas regras são um conjunto de ações pré-computadas para várias situações previstas, simplificando o mapeamento percepção-ação.

3.1.2 Agentes Reativos baseados em modelos

Comparando com o agente reativo simples, o agente baseado em modelo possui um subsistema de processamento de informação a mais, ou seja, *próximo*.

Isto implica em realizar uma nova decomposição no subsistema de tomada de decisão do agente padrão. Em termos de funcionamento esta decomposição equivale à decomposição do passo (3) do ciclo de operação do agente puramente reativo descrito na seção anterior. Mais especificamente, depois do subsistema de percepção ***Ver*** mapear um estado do ambiente em ***S*** em uma percepção em ***P***, um subsistema de atualização de estado interno, ***Próximo : IxP*** → ***I***, mapeia a percepção em ***P*** e o estado interno corrente em ***I*** = $\{i_1, ..., i_m\}$, em um novo estado interno; que, por sua vez, é processado pelo subsistema de tomada de decisão, ***Ação: I*** → ***A***, para selecionar uma ação possível em ***A*** [GONÇALVES, 2009].

O agente possui em uma estrutura de dados interna uma descrição do estado interno do ambiente onde é acessada nas mudanças que ocorrem em seu ambiente. Esta arquitetura é uma melhoria do agente reativo simples através do seu histórico de ações.

3.1.3 Agentes baseados em objetivos

Muitas vezes além do estado em que o agente se encontra é necessário traçar uma meta a se alcançar, onde este agente pode tomar decisões para alcançar algum objetivo. É preciso que o agente tenha informações sobre os objetivos, no caso, situações desejáveis para agir. Este comportamento pode ser alterado na medida em que percebe alguma alteração do ambiente.

Busca e planejamento dedicam-se a encontrar sequências de ações que alcançam os objetivos do agente [RUSSELL; NORVIG, 2004]. Para Silva [2004], o planejamento é uma sequência de ações que levam o agente a atingir um objetivo. Russel e Norvig [2004] consideram que esta sequência de ações pode ser resultante do processo de busca.

3.1.4 Agentes baseados na utilidade

Muitas vezes os objetivos produzem uma distinção binária de “satisfaz” e “não satisfaz”. É preciso que o agente tenha neste caso, é interessante que sobre os objetivos haja uma medida de desempenho como “mais rápido”, “mais seguro”, “mais econômico” etc. A esta característica é dada o nome de utilidade que serve como um requisito funcional ao estado interno do agente, principalmente se o ambiente é parcialmente observável.

A utilidade é útil quando objetivos são contraditórios, escolhendo o objetivo mais adequado, ou quando há vários objetivos a serem alcançados, mas há poucas chances de acontecer, ela fornece um meio pelo qual esta probabilidade pode aumentar.

Um agente orientado a utilidade é mais eficaz em duas situações: 1) quando há incerteza no resultado das ações, e 2) quando há mais de um objetivo a ser alcançado [GONÇALVES, 2009]. A utilidade fornece um meio pelo qual a probabilidade de sucesso pode ser ponderada em relação à importância dos objetivos [RUSSELL; NORVIG, 2004]. Em contrapartida, esta análise envolve um maior tempo de processamento por parte do agente.

3.2 MAS-ML 2.0

MAS-ML é uma linguagem de modelagem que estende a UML para permitir a modelagem das entidades que comumente fazem parte de SMAs, e seus relacionamentos. A extensão à UML é baseada nos conceitos definidos pelo *framework* conceitual TAO [SILVA; CHOREN; LUCENA, 2007], que define as entidades: agente, papel de agente, objeto, papel de objeto, organização e ambiente.

MAS-ML 2.0 [GONÇALVES, 2009] é uma evolução da linguagem de modelagem MAS-ML visando adequá-la para modelagem de arquiteturas internas de: (i) agentes reativos simples; (ii) agentes reativos baseados em conhecimento; e (iii) agentes baseados em objetivo com planejamento e (iv) agentes baseados em utilidade. Em termos práticos, a extensão da linguagem envolveu a evolução do metamodelo de MAS-ML com o objetivo de inserir novas meta-classes (*AgentPerceptionFunction* e *AgentPlanningStrategy*), associar estereótipos à meta-classe *Agent* e especificar como os relacionamentos, previamente definidos, interagem com os demais elementos do metamodelo. O metamodelo conceitual da linguagem MAS-ML 2.0 é apresentado na Figura 1, enquanto que a Tabela 2 mostra as características internas dos agentes em cada arquitetura.

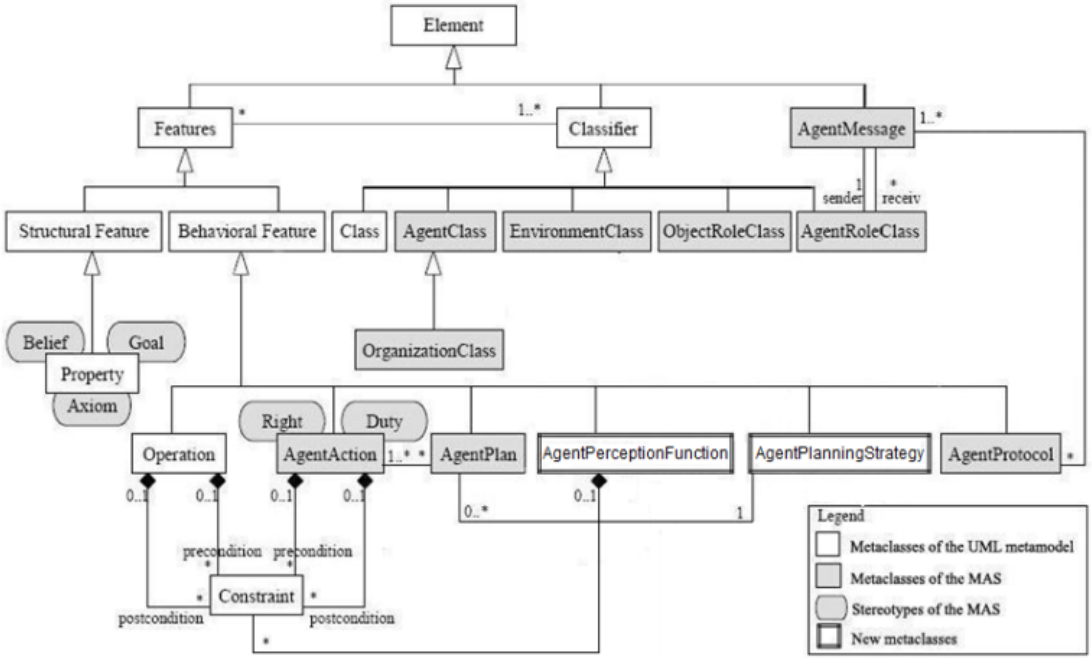


Figura 1 O metamodelo da linguagem MAS-ML 2.0 [GONÇALVES, 2009]

Tabela 2 Características de agente em MAS-ML 2.0 [GONÇALVES, 2009]

Arquitetura interna	Características Estruturais / Mentais	Características Comportamentais	Modelagem
Agente Reativo Simples	-	Percepção e Ação (guiada por Regras Condição-Ação)	MAS-ML 2.0
Agente Reativo baseado em Conhecimento	Crença	Percepção, Função próximo e Ação (guiada por Regras Condição-Ação)	MAS-ML 2.0
Agente baseado em objetivo com plano	Objetivo e Crença	Plano e Ação (baseados na seleção do plano de acordo com o objetivo)	MAS-ML
Agente baseado em objetivo com planejamento	Objetivo e Crença	Percepção, Função próximo, Função de formulação de objetivo, Função de formulação do problema, Planejamento e Ação	MAS-ML 2.0
Agente baseado em Utilidade	Objetivo e Crença	Percepção, Função próximo, Função de formulação de objetivo, Função de formulação do problema, Planejamento, Função Utilidade e Ação	MAS-ML 2.0

Exceto o agente baseado em objetivo com plano [SILVA, 2004], os outros agentes foram baseados nas arquiteturas internas definidas em Russell e Norvig [2004].

Em consistência com as estruturas de agente, o papel de agente foi estendido em MAS-ML 2.0 para se adequar ao tipo de agente ao qual ele é associado. Desta

forma, três tipos de papel de agente foram definidos: papel de agentes reativos simples, papel de agentes reativos baseados em conhecimento e papel de agentes proativos. O papel de agente reativo não possui crenças e objetivos e é exercido pelo agente reativo simples. No papel de agente reativo baseado em conhecimento é acrescentado a propriedade estrutural crença e é utilizado pelo agente reativo baseado em conhecimento. Por último, o papel de agente proativo possui as propriedades estruturais de crença e objetivo e é exercido pelos agentes proativos, a saber: agente baseado em objetivo com plano, agente baseado em objetivo com planejamento e agente baseado em utilidade.

Através de MAS-ML, a MAS-ML 2.0 disponibiliza vários tipos de relacionamentos entre as entidades para contemplar as diferentes interações entre elas. Resumidamente, as características de cada uma são listadas a seguir de acordo com [SILVA, 2004]:

- *Inhabit*: O relacionamento *inhabit* especifica que a instância de entidade que reside – o cidadão – é criada e destruída no habitat e, portanto, pode entrar e sair dele, respeitando suas permissões. Um cidadão não pode residir em dois habitats ao mesmo tempo. O *habitat* conhece todos os cidadãos que residem nele, e cada cidadão conhece seu *habitat*. Esse relacionamento aplica-se a ambientes e agentes, a ambientes e objetos e a ambientes e organizações;
- *Ownership*: O relacionamento *ownership* especifica que uma entidade – o membro – é definida no escopo de outra entidade – o proprietário – e que um membro deve obedecer a um conjunto de restrições globais definidas pelo proprietário. O membro não existe fora do escopo de seu proprietário. Os proprietários conhecem seus membros, e cada membro conhece seu proprietário. Esse relacionamento aplica-se a classes de papel – os membros – e às classes de organização – os proprietários;
- *Play*: O relacionamento *play* especifica que uma entidade está relacionada a um papel. Quando uma classe de entidade está relacionada a uma classe do papel pelo relacionamento *play*, isso significa que uma instância de entidade pode exercer uma instância de papel. Uma instância do agente

e de uma suborganização deve executar pelo menos uma instância de papel;

- *Specialization/Generalization*: O relacionamento *specialization* define que a subentidade que especializa a superentidade herda as propriedades e os relacionamentos definidos na superentidade. As propriedades herdadas também podem ser redefinidas pela subentidade. Ademais, a subentidade também pode definir novas propriedades e novos relacionamentos. O relacionamento *specialization* definido pelo metamodelo de UML foi estendido para ser aplicado a todas as entidades do TAO;
- *Control*: O relacionamento *control* define que a entidade controlada deve fazer tudo que a entidade do controlador pedir. A entidade do controlador conhece as entidades controladas, e cada entidade controlada conhece as entidades que a controlam. O relacionamento *control* pode ser usado entre dois papéis do agente. As entidades controladas e do controlador serão os agentes ou as organizações que estão exercendo os papéis;
- *Dependency*: Esse relacionamento define que uma entidade – o cliente – pode ser definida para depender de outra – o fornecedor – para realizar seu trabalho. Uma dependência é um relacionamento de uso que especifica que uma alteração na especificação de algo (fornecedor) pode afetar outra coisa usada por ele (cliente), mas não necessariamente o contrário. O cliente sempre conhece seus fornecedores, mas o contrário pode não ser verdadeiro;
- *Association*: Uma associação especifica um relacionamento de semântica que pode ocorrer entre instâncias tipificadas. Se uma entidade estiver associada à outra entidade, ela saberá da existência da outra entidade e, então, poderá interagir com ela;
- *Aggregation*: Se uma entidade estiver agregada à outra, dizemos que ela é o agregador de partes. O relacionamento *aggregation* define a entidade que é o agregador e a entidade que é a parte. O agregador pode usar a funcionalidade disponível em suas partes para realizar seu trabalho. A parte define um conjunto de funcionalidades usadas pelo agregador. Esse

relacionamento foi estendido para ser aplicado entre papéis de objeto e entre papéis do agente;

Em relação aos diagramas, o metamodelo de MAS-ML na sua concepção original [SILVA, 2004] contempla os diagramas de Classes, Organização, Papéis, Sequência e Atividades. MAS-ML 2.0 manteve as representações existentes, no entanto, esses diagramas foram alterados adequando o metamodelo original da linguagem para a modelagem das diferentes arquiteturas internas de agentes.

O diagrama de organização modela as organizações do sistema, identificando seus ambientes e regras. Este diagrama mostra os relacionamentos: Posse, Exerce e Habita. Já o diagrama de papéis é responsável por explicitar os relacionamentos entre os papéis de agente e os papéis de objeto. Este diagrama mostra os relacionamentos controle, dependência, associação, agregação e especialização.

No intuito de auxiliar a representação do diagrama de organização de MAS-ML 2.0, a ferramenta MAS-ML *Tool* [FARIAS et al., 2009], [GONÇALVES et al., 2010a] e [GONÇALVES et al., 2010b], possibilita a criação deste tipo de diagrama, com suas entidades e relaciomentos definidos pela linguagem de modelagem. Nesta dissertação será utilizado apenas o digrama de organização para modelar o ambiente, organizações, agentes e papéis de agente e os relacionamentos posse, exerce e habita pertencentes a estas entidades.

Mais detalhes sobre representações das entidades, relacionamentos e diagramas de MAS-ML 2.0 são encontrados em Silva, Choren e Lucena [2005], Silva, Choren e Lucena [2007] e Gonçalves [2009].

3.3 JADE

JADE é um *middleware* para o desenvolvimento de sistemas multi-agentes que inclui:

- Um ambiente de execução onde os agentes JADE "vivem" e que deve estar ativo em um *host* antes que um agente possa ser executado;
- Uma biblioteca de classes que programadores podem usar para desenvolver agentes;

- Um conjunto de ferramentas gráficas que permite a administração e o monitoramento das atividades dos agentes em execução;

Por utilizar uma linguagem orientada a objetos (Java), JADE possui uma forma simples de especificação de agentes e comunicação entre os mesmos. Sua estrutura abrange os seguintes aspectos:

- *Container*: Instância de um ambiente JADE, onde os agentes executam. Ao iniciar o JADE, um *main container* é criado;
- Plataforma: Conjunto de contêineres ativos;
- Suporte a ontologias: Definição de vocabulários;
- *Directory Facilitator* (DF): Serviço de páginas amarelas na plataforma;

A plataforma é utlizada para dar suporte aos contêineres, ela funciona de forma similar ao ambiente de MAS-ML 2.0. A ideia de contêiner segue o raciocínio de organização, onde os agentes residem e podem transitar. Ao dar início a uma plataforma, um contêiner principal é criado automaticamente e dois agentes implícitos, AMS e DF, são também automaticamente criados, os quais possuem um papel importante no funcionamento do sistema. Devido à exsitência de vários componentes em JADE, isto possibilita a distribuição dos mesmos em diferentes máquinas [BELLIFEMINE; CAIRE; GREENWOOD, 2007].

O AMS (*Agent Management System*) é um agente que exerce o controle sobre o acesso e o uso da plataforma que mantém a lista de identificadores dos agentes criados na plataforma. Só existe um AMS por plataforma que reside no contêiner principal. Os agentes criados deverão se registrar no AMS.

O DF (*Directory Facilitator*) é um agente que oferece o serviço das páginas amarelas, que visa classificar os serviços prestados pelos agentes, ou seja, os classificados. É aqui onde o agente disponibiliza seus serviços de forma a poder ser achado e estabelecer a comunicação com outros agentes. Assim como o AMS, só existe um DF por plataforma, porém o registro dos agentes no DF é facultativo, visto que vai depender da necessidade do agente pelo serviço de publicação de seus serviços nas páginas amarelas ou não.

As classes de agentes em JADE herdam diretamente ou indiretamente da classe *jade.core.Agent*. Por exemplo, a Figura 2 mostra a codificação do agente como uma classe *Agent2* que herda da classe *jade.core.Agent*.

A classe *Agent* representa uma classe base comum para a definição de agentes por parte do usuário. Portanto, do ponto-de-vista do programador, um agente JADE é simplesmente uma instância de uma classe Java que herda da classe *Agent*. Isto implica na herança de características para suportar interações básicas com a plataforma de agentes (registro, configuração e gerenciamento remoto).

Um conjunto básico de métodos é chamado para codificar comportamentos personalizados do agente, como métodos para enviar ou receber mensagens, utilizar protocolos de comunicação padrão, registro de vários domínios, entre outros.

O ciclo de vida do agente é definido de acordo com o padrão FIPA [FIPA, 2011]. O primeiro passo consiste na execução do construtor do agente, seguido da atribuição de um identificador para o agente a ser inserido no sistema. O método *setup* (veja Figura 2) é executado a partir do momento em que o agente inicia suas atividades. Para atribuir um comportamento específico ao agente, o método *setup* deve ser sobrescrito.

```
public class Agent2 extends Agent {
    @Override
    protected void setup() {
        super.setup();
        addBehaviour(new ExemploComportamento());
    }
}
```

Figura 2 Implementação e atribuição de comportamento de agentes em JADE.

Uma característica importante do JADE é a definição dos comportamentos que conterão as ações dos agentes. A classe principal desta abordagem é a *jade.core.behaviours.Behaviour*, a partir da qual são definidas várias outras subclasses que servirão para um propósito específico como composição de outros comportamentos ou duração das tarefas. Uma classe de comportamento define basicamente dois métodos: *action()* e *done()*, ver Figura 3. A classe deve manter o estado da execução de modo que o método *done()* retorne um valor lógico falso (equivalente ao literal *false* em Java) onde ele informar o término da execução do

método *action()*. Caso contrário, deve retornar um valor lógico verdadeiro (equivalente ao literal *true* em Java). A Figura 2 mostra como atribuir comportamento a um agente JADE, enquanto que a Figura 3 mostra um exemplo de comportamento sobrescrevendo os métodos *action()* e *done()* da classe base *Behaviour*.

```
class ExemploComportamento extends Behaviour {
    @Override
    public void action() {
        //Aqui deve ser implementado um comportamento para o agente
    }

    @Override
    public boolean done() {
        //Retorna verdadeiro caso seja necessário executar
        //o método action novamente.
        return false;
    }
}
```

Figura 3 Codificação do comportamento do agente em JADE.

Os agentes podem se comunicar entre si enviando mensagens assíncronas, pois eles podem tomar a decisão de resposta. A comunicação segue o padrão FIPA ACL (*Agent Comunication Language*) [FIPA, 2011] e podem ser definidas ontologias que servirão como o vocabulário para cada situação que o agente se encontrar.

Nesta seção foram apresentadas as funcionalidades principais do JADE, mas há outras classes que servem de apoio nas funcionalidades do SMA a ser desenvolvido e isto pode ser visto na documentação do mesmo. Há algumas classes adicionais em JADE, as quais podem ser estendidas que servirão para tratar elementos mentais de agentes como: crença, objetivos, ações, planos etc. Estas classes que compõem estes elementos serão vistas no próximo capítulo.

3.4 Desenvolvimento orientado a modelos

O desenvolvimento orientado a modelos, ou *model-driven development* (MDD), é uma abordagem de desenvolvimento de software onde o foco é colocado na modelagem e não na implementação [FRANCE; RUMPE, 2007]. Neste contexto, o código é gerado a partir dos artefatos de modelagem. Esta abordagem prevê a automação através da execução de modelos, transformações e técnicas de geração de código.

No esforço pela padronização com vistas a facilitar a integração de recursos de TI (Tecnologia da Informação), a OMG[2] (*Object Management Group*) forneceu uma arquitetura orientada a modelos, MDA (*Model Driven Architecture*). Esta arquitetura é baseada em um conjunto de padrões, tais como UML junto com MOF (*Meta Object Facility*), XMI (*XML Metadata Interchange*) e CWM (*Common Warehouse Metamodel*) que disponibiliza alguns modelos de núcleo ou perfis para desenvolvimento [OMG, 2011].

Seguindo o contexto de MDA, geradores de código são ferramentas que, a partir de um conjunto de instruções de entrada, oferecem como saída código fonte em uma ou mais linguagens de escolha do usuário [LUCAS, 2000]. O exemplo mais tradicional que se tem é o compilador que transforma um código escrito através de alguma linguagem de programação de alto nível para código de máquina.

Os tipos de geração de código [LUCAS, 2000] podem ser divididos nas seguintes categorias:

- Baseados em *templates*: recebem como entrada um ou mais *templates*, consistindo em arquivos com algumas informações requeridas para formatar o texto escrito em outro texto;
- Baseados em banco de dados: é criado um banco de dados relacional e a partir deste, as classes são geradas, mas é mais voltado para gerar entidades-relacionamento;
- Orientados a modelos: recebem como entrada um conjunto de diagramas. Usando este tipo de gerador facilita também a documentação, pois a mesma pode ser modelada;
- Mista: são geradores que misturam as características de duas categorias ou mais.

O desenvolvimento com base em modelos se apresenta como uma abordagem para auxiliar na geração de código, pois o código pode ser gerado diversas vezes sem comprometer o modelo [SOUZA, 2011]. A OMG definiu padronizações deste processo que não obrigatoriamente são atrelados a uma plataforma específica, ou

[2] OMG (*Object Management Group*) é uma organização que promove padrões de desenvolvimento visando facilitar o desenvolvimento des aplicações orientadas a objetos. http://www.omg.org/.

seja, os conceitos podem ser aplicados a diferentes linguagens de modelagem e implementação. O processo de transformação ocorre através de etapas propostas pela OMG. Os artefatos gerados em cada uma destas etapas possuem um nível de independência para um melhor aproveitamento, como exemplo, a mesma aplicação em diferentes linguagens de implementação. Os conceitos de cada uma destas fases estão descritas a seguir [BEYDEDA; BOOK; GRUHN, 2005] e ilustradas de acordo com a Figura 4:

- CIM (*Computation Independent Model*): voltado apenas para compreensão das regras de requisitos para especificar o domínio da aplicação a se definir, serviços e entidades envolvidos. Especifica o domínio, mas não o funcionamento interno. Ex.: Requisitos funcionais da aplicação a ser desenvolvida;
- PIM (*Platform Independent Model*): explicita relações entre propriedades de uma entidade ou interações entre entidades. Define a arquitetura do projeto, mas sem detalhes de implementação. Ex.: modelos em MAS-ML 2.0, projetadas em MAS-ML *Tool*;
- PSM (*Platform Specific Model*): aqui especifica os detalhes como o sistema trabalhará em uma plataforma específica. Ex.: Java;
- PDM (*Platform Definition Model*): Tendo a plataforma escolhida (PSM), é necessário verificar como a mesma funciona e os serviços que oferece para que as definições estabelecidas no PIM possam ser realizadas. São os detalhes PSM. Ex.: *plugin* Acceleo;

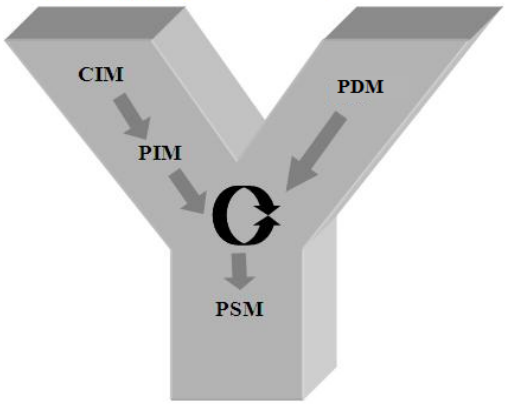


Figura 4 Processo MDA

Fonte: http://research.petalslink.org/pages/viewpage.action?pageId=3639295

Há ainda outro conceito estabelecido pelo MDA, o ISM (*Implementation Specific Model*), o qual é o resultado propriamente dito, ou seja, é o código ou produto final gerado a partir da transformação do modelo da aplicação.

Dentre as ferramentas de suporte a MDA podemos destacar:

- AndroMDA [ANDROMDA, 2011]: recebe um XMI como entrada. Possui uma série de *templates*, respectivos para cada linguagem que se quer gerar. Pode ser integrada com outros ambientes de modelagem ou importar outros modelos armazenados no formato XML. Há possibilidade de estender os *templates*;
- Acceleo [ACCELEO, 2011]: permite a geração automática de código a partir de um modelo UML2, Ecore ou um metamodelo definido pelo usuário que esteja de acordo com o EMF(*Eclipse Modeling Framework*). Compatível com XML 1.0 e 2.0. Já possui módulos para geração Java, C, Python, PHP, Hibernate.

Alguns critérios que podem ser utilizados para a análise e escolha entre os mecanismos de suporte a MDA segundo Maia [2006] e Souza [2009] são:

- Integração com Eclipse: permite integrar com a ferramenta MAS-ML *Tool*;
- Documentação: indica que a ferramenta utilizada possui boa documentação para consultas a tutoriais;
- Possibilidade de extensão: indica se a abordagem permite a sua extensão para a definição de novas transformações de natureza não prevista pela abordagem.
- Transformação modelo-texto: indica se a abordagem apóia a definição de transformações do tipo modelo-texto;

De acordo com as abordagens, temos o seguinte comparativo (ver Tabela 3) das ferramentas em MDA, onde “sim” atende, “não” não atende e “?” não foi encontrado na literatura, [MAIA, 2006] e [SOUZA, 2009]:

Tabela 3 Tabela Comparativa das ferramentas

Critérios	Abordagens	
	AndroMDA	Acceleo
Integração com Eclipse	Sim (através de outros	Sim

	plugins)	
Transformação modelo-texto	Sim	Sim
Documentação	Sim	Sim
Possibilidade de extensão	Sim	Sim

Estas duas ferramentas de geração de código oferecem suporte à geração de código a qualquer dos *frameworks* relacionados neste capítulo. AndroMDA não possui compatibilidade com nenhuma ferramenta que faça uso de UML 2.0 e também não possui uma integração (*plugin*) direta com a ferramenta Eclipse, neste caso, é necessário utilizar outros *plugins* para esta adaptação. Por esta razão, o Acceleo foi escolhido para atender as necessidades deste trabalho.

4 EXTENSÃO DE JADE PARA MAS-ML 2.0

Considerando os recursos oferecidos a nível de modelagem pela linguagem MAS-ML 2.0, dentre eles as arquiteturas internas de agentes inteligentes, torna-se necessária a incorporação de algumas classes Java em JADE, transferindo a informação presente na modelagem para atributos e métodos nas novas classes. O mapeamento entre os conceitos de modelagem e de implementação consiste em identificar quais elementos do metamodelo conceitual referente às entidades de MAS-ML 2.0 existem no *framework* JADE. A extensão do *framework* obtida é denominada de JAMDER (*JADE to MAS-ML 2.0 Development Resource*) e o detalhamento é descrito a seguir.

4.1 Agentes

A linguagem MAS-ML 2.0 contempla a modelagem de agentes com diferentes propriedades estruturais e comportamentais, definidas no metamodelo da linguagem. O primeiro passo do mapeamento é identificar quais elementos do metamodelo conceitual de MAS-ML 2.0 referente a agentes existem no *framework* JADE.

De acordo com a definição de MAS-ML 2.0 existem cinco diferentes tipos de agentes, a saber: (i) agentes reativos simples, (ii) agentes reativos baseados em conhecimento, (iii) agentes baseados em objetivos com plano, (iv) agentes baseados em objetivos com planejamento e (v) agentes baseados em utilidade, ver Figura 5. Levando em conta os aspectos comuns e também as diferenças no nível de modelagem, três hierarquias de classes para agentes podem ser estabelecidas: uma para MAS-ML (agente baseado em objetivo com plano), outra para agentes reativos de MAS-ML 2.0 (agente reativo simples e agente reativo baseado em conhecimento) e outra para agentes cognitivos ou proativos de MAS-ML 2.0 (agente baseado em objetivo com planejamento e agente baseado em utilidade). Esta classificação foi necessária para agrupar agentes que possuem características em comum, buscando a reutilização através de herança.

Considerando que todos os agentes têm como propriedades comuns estar em um ambiente e executar pelo menos um papel nas organizações de que fazem

parte, a classe denominada *jamder.agents.GenericAgent*, a qual herda de *jade.core.Agent*, foi definida para representar estas propriedades e demais atributos comuns entre as três ramificações da hierarquia de agentes especificada. Esta classe possui um construtor que recebe o nome do agente, o ambiente e o papel de agente inicial. Os atributos comuns definidos na classe *GenericAgent* são: (i) a lista de papéis de agente, (ii) a lista de organizações das quais o agente pode participar, (iii) o ambiente em que o agente se encontra (iv) e a lista de ações que ele pode executar. Estas listas são instâncias da classe *java.util.Hashtable* e são protegidas *(protected)* para garantir acesso somente pelas subclasses, exceto a lista de papéis de agente. Todas as listas definidas possuem cinco métodos de acesso no seguinte padrão:

- *getXXX(String name)* – retorna a instância de um elemento da lista a partir do nome;
- *addXXX(name, XXX)* – adiciona o elemento XXX identificado pelo nome;
- *removeXXX(name)* – remove o elemento da lista a partir do nome e retorna a lista;
- *removeAllXXX()* – remove todos os elementos da lista;
- *getAllXXX()* – retorna a lista de elementos XXX;

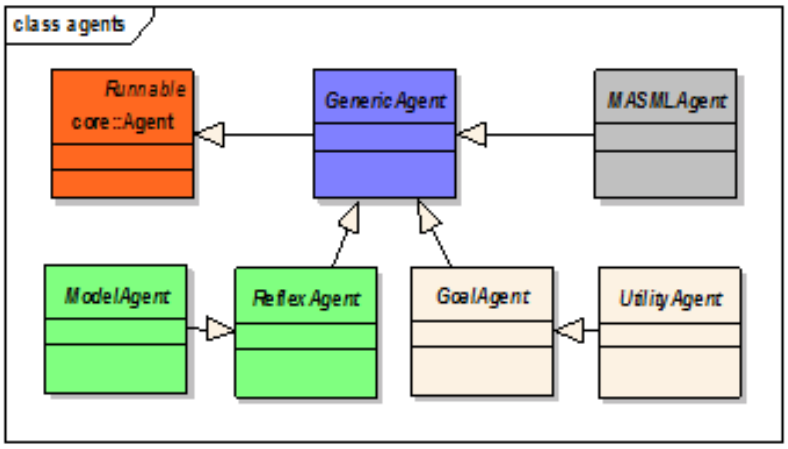


Figura 5 Hierarquia de Agentes em JAMDER

Em Silva [2004], o agente possui pelo menos um papel de agente que contém crenças, objetivos e ações. Com a versão 2.0 mais duas representações de papéis de agente foram acrescentadas. Seguindo este raciocínio, os papéis de agente

também foram organizados em três categorias para atender às três hierarquias dos agentes, o detalhamento da hierarquia de papéis é apresentado mais adiante. Ao adquirir o papel, o agente adquire também as crenças e os objetivos do papel, caso haja crenças ou objetivos. O agente somente poderá executar suas ações, somente se, as mesmas ações também forem definidas no papel de agente que ele exerce.

4.1.1 Agente Reativo Simples

De acordo com Gonçalves [2009], as regras possuem a seguinte forma: “se condição então ação”, e determinam a ação a ser executada se a percepção ocorrer (condição-ação). A classe criada para representar este agente é a *jamder.agents.ReflexAgent*, a qual herda de *GenericAgent*. A classe *ReflexAgent* contém um lista de regras condição-ação do tipo *Hashtable*<*String, String*>, onde o primeiro elemento representa o nome da percepção e o segundo o identificador da ação correspondente.

Para representar a característica de percepção, foi criada a classe chamada *jamder.behavioural.Sensor*, a qual herda de *jade.core.behaviours.TickerBehaviour*, pois, de tempos em tempos, realiza a captura das percepções do ambiente, as quais são tratadas pelo método abstrato *percept* (*perception*). O intervalo de tempo deve ser definido pelo desenvolvedor na classe *Sensor*. À medida que o *Sensor* (comportamento) percebe algo, o agente é informado e, caso haja equivalência, a ação correspondente é executada pelo agente.

4.1.2 Agente Reativo Baseado em Conhecimento

Este agente armazena um histórico, conhecimento ou crença para sua tomada de decisão e as crenças são representadas através de uma lista da classe *jamder.structural.Belief* na forma de atributo deste agente, representado pela classe *jamder.agents. ModelAgent*. Esta classe herda da classe *ReflexAgent*, incorporando adicionalmente o método abstrato *nextFunction(belief, perception)* que, nas classes concretas, deve conter a implementação responsável por mapear as percepções e o estado interno (representação do ambiente ou modelo) atual para um novo estado interno, e será utilizado para selecionar a próxima ação [GONÇALVES, 2009].

Partindo da premissa de que as crenças não podem ser acessadas por outros agentes, ou seja, são privadas ao próprio agente, i.e., a classe concreta que herda

de *ModelAgent* é que tem esta capacidade de mudar suas crenças, seus métodos de acesso precisam ser definidos como *protected*.

Este tipo de agente, ao assumir o papel, irá incorporar todas as crenças que estiverem no papel. Caso haja crenças iguais, as crenças do agente serão substituídas pelas do papel. O método *addAgentRole(name, role)* de *ModelAgent* é sobrecarregado para incorporar este princípio. Como os objetivos não fazem parte da estrutura destes agentes, então o papel de agente também não possui este componente.

4.1.3 Agente Baseado em Objetivo com Planejamento

O agente baseado em objetivo com planejamento tem como capacidade principal, gerar um plano em tempo de execução, onde os planos são compostos sequencialmente por ações. Este tipo de agente também possui um conjunto de objetivos, que consistem em uma meta que o agente deseja alcançar.

A classe que implementa este agente é a *jamder.agents.GoalAgent* que herda de *GenericAgent,* mas incorpora três métodos abstratos adicionais: a função de formulação de objetivo que recebe o estado atual (crença ou modelo) e retorna um objetivo formulado, a função de formulação de problema que identifica as ações necessárias para o estado e objetivo, e o método de planejamento que retorna a sequência de ações. Além disso, este agente possui percepções e a função próximo que foram inclusos nesta hierarquia através da lista *perceives* e do método *nextFunction(belief, perception)*, respectivamente.

Ao adquirir o papel, o agente baseado em objetivo irá incorporar também os objetivos do papel. Caso haja objetivos iguais, os objetivos do agente serão substituídos pelos do papel. O método *addAgentRole(name, role)* desta classe é sobrecarregado para incorporar este princípio.

O método abstrato da função de formulação de objetivo, representado por *formulateGoalFunction(belief)*, recebe uma crença e retorna o objetivo formulado. O algoritmo que implementa este método é fornecido pelo desenvolvedor.

O método abstrato da função de formulação de problema, representado por *formulateProblemFunction(belief, goal)*, recebe a crença e o objetivo e retorna o

problema que será usado no planejamento. O problema consiste de um subconjunto de ações das ações do agente [RUSSELL; NORVIG, 2004].

Enfim, o método de planejamento, *planning(actions)*, recebe o problema (lista de ações) e retorna a lista sequencial destas ações a serem seguidas para atingir o objetivo proposto. De acordo com Gonçalves [2009], o planejamento é realizado com base nas ações disponíveis na lista de ações, no intuito de criar uma sequência de ações (plano).

Assim como a hierarquia dos agentes reativos, esta hierarquia de agentes também necessita de um sensor para tratar as percepções do ambiente. A fim de contemplar este propósito, há a definição do método *percept (perception)* neste agente que utiliza o comportamento da classe *Sensor*.

4.1.4 Agente Baseado em Utilidade

O agente baseado em utilidade possui a mesma estrutura do agente baseado em objetivo com planejamento, porém, considerando que seu planejamento pode estar ligado a mais de um objetivo a ser atingido, é adicionada a função utilidade com o intuito de guiar o comportamento do agente. Considerando que estes objetivos podem ser conflitantes, a função utilidade é responsável por determinar o grau de utilidade em relação aos objetivos associados. A classe que representa este tipo de agente é a *jamder.agents.UtilityAgent* e herda de *GoalAgent*.

A classe *UtilityAgent* contém o método abstrato *utilityFunction(action)* usado para identificar a prioridade das ações em relação aos objetivos do agente, tendo como base a ação recebida. Este método é chamado várias vezes durante a formulação do problema para identificar as ações que serão usadas. O retorno deste método consiste em um número que indica a prioridade da ação.

4.1.5 Agente Baseado em Objetivo com Plano

O agente baseado em objetivo com plano herda diretamente de *GenericAgent,* uma vez que incorpora os atributos de crenças, e objetivos além dos planos e ações, mas não contém as funções de formulação de objetivo, formulação de problema, função próximo, função utilidade e percepções definidas nos outros agentes. Uma

diferença importante é que seu plano é definido em tempo de modelagem. A classe que representa é a *jamder.agents.MASMLAgent*.

4.1.6 Características estruturais

A verificação das classes JADE que podem ser utilizadas para representar as propriedades dos agentes é imprescindível. Para tanto, é preciso verificar os conceitos de cada uma das características estruturais dos agentes, a saber: crenças (*Belief*) e objetivos (*Goal*).

Em JADE não foram encontradas propriedades correlativas para crenças e objetivos. A estrutura entre crença e objetivo é similar (cada um deles é composto pelos campos: nome, tipo e valor) e são propriedades de todos os agentes, exceto o agente reativo simples e o reativo baseado em conhecimento. Para atender a essa estrutura, foi criada a classe *jamder.structural.Property* que contém estes campos e são herdados pelas classes *jamder.structural.Belief* e *jamder.structural.Goal*, as quais representam as crenças e os objetivos, respectivamente.

Considerando que os objetivos podem ser simples ou compostos, as classes *jamder.structural.LeafGoal* e *jamder.structural.CompositeGoal*, subclasses de *Goal*, representam estas características, respectivamente. Além disso, a classe *Goal* contém uma lista dos planos que são associados a um objetivo e o atributo booleano *achieved* informa se o objetivo foi alcançado ou não.

4.1.7 Características comportamentais

As características comportamentais dos agentes são representadas através de: planos (*Plan*), mensagens (*ACLMessage*), ações (*Action*) e suas condições (*Condition*).

Ações são características comportamentais de agentes [SILVA, 2004], e, portanto devem ser implementadas em relação à classe de JADE, *jade.core.behaviours.Behaviour*. Uma classe de *Behaviour*, chamada *jamder.behavioural.Action*, foi criada para este propósito. A lista de ações do agente indica que o agente está apto a realizar tais ações, porém, para que sejam de fato executadas pelo agente, é necessário chamar o método *addBehaviour(action)* de JADE, passando como parâmetro uma das ações contidas na lista.

As pré-condições definem como o ambiente precisa estar para que a ação possa ser executada. Em contrapartida, as pós-condições definem as condições que irão se tornar verdadeiras quando a execução da ação for concluída com sucesso. Neste sentido, a classe *Action* define dois atributos: a lista de pré-condições e a lista de pós-condições, ambas representadas por uma nova classe chamada *jamder.behavioural.Condition*, definida como subclasse de *Property*. A classe *Action* possui um método chamado *execute()*, responsável por implementar as ações concretas de *Action*. As pré-condições e as pós-condições, por sua vez, possuem cinco métodos de acesso, de maneira análoga às listas em *GenericAgent*.

Cada plano define um conjunto de ações e uma ordem de execução, que em JADE é implementada pela classe *jade.core.behaviours.SequentialBehaviour*. Similarmente, foi criada em JAMDER uma classe chamada *jamder.behavioural.Plan* que herda de *SequentialBehaviour*, pois a cada ação adicionada, a execução ocorre de forma sequencial. A classe criada inclui dois atributos adicionais, o objetivo associado e uma lista de ações, cuja ordem básica no plano é determinada pelo método abstrato *execute()*. As ações do agente que compõem o plano têm que corresponder com as ações definidas no papel de agente, ou seja, as ações devem estar tanto no agente quanto no papel que este agente exerce.

Os agentes se comunicam entre si através de mensagens, as quais são armazenadas em uma fila. O agente decide o momento de fazer a leitura de forma assíncrona. Segundo Silva [2004], uma mensagem é definida por um rótulo que especifica o tipo da mensagem, seu conteúdo, emissor (agente responsável por enviar a mensagem) e o destinatário. Em JADE há uma classe correspondente chamada *jade.lang.acl.ACLMessage* [FIPA, 2011] capaz de implementar este comportamento.

Como todos os tipos de agente propostos herdam de *jade.core.Agent*, esta classe possui uma fila de mensagens recebidas do tipo *ACLMessage*, e como consequência desta herança, esta fila é herdada pelos agentes do *framework* JAMDER. As mensagens na fila de recebimento ficam armazenadas até que a mensagem seja lida, e pode ser recuperada através do método *receive()* de *Agent*.

MAS-ML define que o agente além possuir uma fila de recebimento de mensagens, também possui uma fila de mensagens enviadas, porém JADE não

mantém um registro de mensagens enviadas. Consequentemente, foi criado um atributo em *GenericAgent* que armazena as mensagens enviadas pelo agente. O método *sendMessage(ACLMessage)* é utilizado para incluir a mensagem no momento em que ela é enviada.

Como nenhum agente pode conhecer a estrutura interna de outro agente, torna-se necessária a existência de um identificador em cada agente para que o envio de mensagens entre si possa ser realizado. O identificador proporcionado por JADE consiste em uma instância interna de *jade.core.AID,* criada automaticamente. Esta instância *AID* contém apenas os elementos necessários para os agentes serem localizados incluindo seus endereços de acesso e nome local. O identificador pode ser recuperado através do método *getAID()* de *Agent*.

4.2 Papel de Agente

MAS-ML 2.0 apresenta três tipos de papéis de agente: (i) papel de agentes reativos simples, (ii) papel de agentes reativos baseados em conhecimento e (iii) papel de agentes proativos. Esta divisão foi necessária, pois um papel de agente adiciona novos objetivos e crenças ao conjunto de crenças e objetivos dos agentes, mas dependendo da arquitetura interna os agentes não possuem algumas destas propriedades. Cada um destes tipos de papel pode ser instanciado ou herdado a fim de ser utilizado.

Cada instância de papel de agente é membro de uma organização, ou seja, ela faz uma aliança entre agente e organização e determina o que o agente pode e deve fazer dentro de uma organização. O papel é exercido por um agente ou por uma suborganização e possui uma identificação que representa esta aliança. Baseado nestas premissas, foi criada a classe chamada *jamder.role.AgentRole* e um conjunto de classes associado ao papel de agente de forma a modelar suas propriedades adequadamente. A classe *AgentRole* representa o tipo de papel de agente e é adequado para os agentes reativos simples devido o mesmo não possuir crenças e objetivos. A Figura 6 representa o conjunto de classes proposto por Silva [2004].

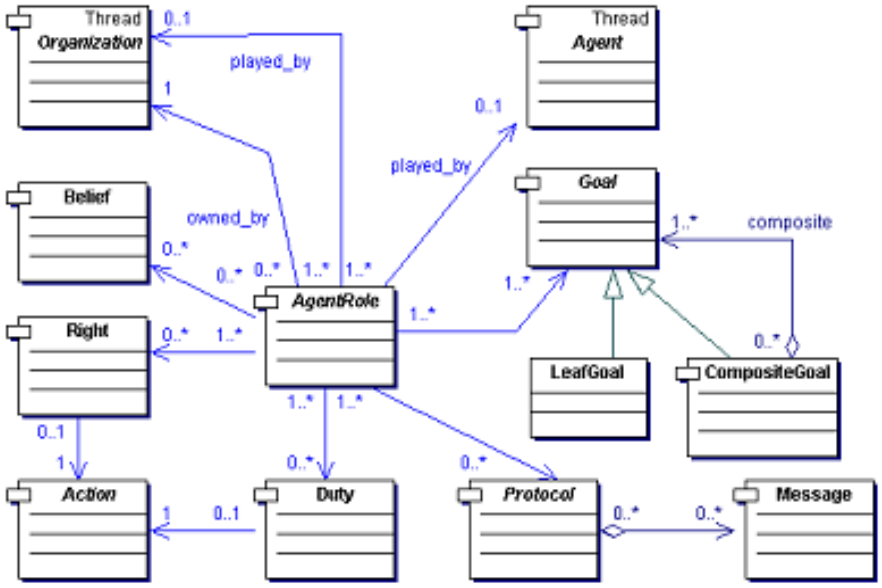


Figura 6 AgentRole e suas classes relacionadas em MAS-ML [SILVA, 2004]

As propriedades básicas dos papéis de agente em um SMA são:

- *owner* – instância da classe *Organization* em que o papel é definido. Esta propriedade pode ser recuperada através do método *getOwner()*. Como o *owner* não muda, então sua definição se dá no construtor da instância do papel de agente;
- *name* – identificador do papel de agente *(String)*, definido e recuperado pelos métodos de acesso *getName()* e *setName(String name)*, respectivamente;
- *player* – indica quem está exercendo o papel. Ele é representado pela classe *GenericAgent*, pois pode ser uma instância de um dos agentes ou uma instância da organização, se estiver exercendo como uma suborganização. Como o *player* também não muda, então sua definição se dá no construtor da instância do papel;

O ciclo de vida de um papel de agente começa quando ele é associado à entidade (agente ou suborganização). Em Silva [2004], este evento é citado como comprometimento e significa que a entidade ativa este papel de agente. Isto significa que a instância do papel de agente nunca é criada sem existir uma instância de agente ou suborganização antes. O papel pode ser cancelado, eliminando o comprometimento da entidade com o papel. Neste caso, é necessário que a

entidade tenha outros papéis a exercer naquela organização em que está sendo cancelado o papel, caso contrário, a instância do agente também é eliminada.

O estado do papel de agente pode ser desativado (inativado ou bloqueado) até que o agente o ative novamente. Para contemplar os status do papel de agente, foi criada a classe *jamder.roles.AgentRoleStatus* definido como *enum* do tipo *boleano*. Os status definidos para o papel de agente são *ACTIVATE*, *DEACTIVATE* e *CHANGE* e podem ser recuperados através do método *getAgentRoleStatus()* de *AgentRole*. O status CHANGE informa que o agente está mudando de um ambiente para outro ou de uma organização para outra. O status CANCEL, definido em MAS-ML 2.0, não é contemplado, visto que ao cancelar (excluir) o papel, a entidade não possui mais a instância do papel.

Os direitos e deveres definidos por um papel de agente estão relacionados às ações que o agente, que está exercendo o papel, deve executar e tem permissão para executar, respectivamente [SILVA, 2004]. Portanto, para representar os direitos e deveres, as classes *jamder.behavioural.Duty* e *jamder.behavioural.Right* foram criadas, onde cada uma identifica apenas um atributo que define as ações associadas a elas. A classe *Action* utilizada pelos agentes também representa a ação associada aos direitos e deveres, onde cada um destes possui apenas uma ação. No papel de agente, os direitos e deveres são definidos como uma instância da classe *Hashtable<String, XXX>,* onde XXX é uma instância de *Duty* ou *Right*.

Quando o agente ou suborganização adquire o papel, as ações de *Duty* e *Right* não são acrescentadas ao agente ou suborganização, mas informam quais podem ou devem ser executadas pelo agente ou suborganização. Como *AgentRole* pode conter várias ações através de *Right* e *Duty*, o método *activeRole()* fornece o serviço de colocar o papel de agente no status ACTIVATE e reinicia as ações do papel de agente. Por outro lado, o método *changeDeactivateRole(AgentRoleStatus)* determina o status do papel dependendo do parâmetro do método e finaliza as ações deste papel. Ao criar uma instância de papel de agente, os direitos e deveres precisam ser analisados através do método *initialize()*. Para sua execução, é necessário que este método seja chamado no construtor da classe que herda de *AgentRole*, direta ou indiretamente.

Um protocolo define o conjunto de mensagens que um agente (ou organização) pode enviar a outros agentes (ou organizações) em uma interação, e as mensagens que pode receber deles [SILVA, 2004]. Em JADE, há vários tipos de protocolo baseado no padrão FIPA para padronizar a “conversação” entre os agentes. Cada uma destas conversações oferece dois tipos de comunicadores: o iniciador, designado ao agente que inicia a comunicação e o participante, que representa o agente que responde ao ato comunicativo iniciado pelo iniciador. Desta forma, em JADE, o protocolo é definido pelo seu tipo e os comportamentos do iniciador e participante. Em JADE, a sequência das mensagens já está predefinida pelo tipo de protocolo, porém o desenvolvedor pode alterar esta sequência reimplementando o protocolo através das classes iniciador e/ou participante escolhidas (ver Tabela 4). Cada protocolo herda direta ou indiretamente de *Behaviour*.

Em MAS-ML 2.0, o papel de agente pode ter vários tipos de protocolo, o que em JADE indica quais tipos de iniciador e/ou participante o papel vai disponibilizar para a entidade. Diante desta premissa, os protocolos são definidos através de uma instância de *Hashtable<String, Behaviour>* em *AgentRole*, onde *Behaviour* representa uma extensão das classes iniciadora ou participante, conforme definidas na Tabela 4:

Tabela 4 Protocolos definidos em JADE.

Protocolo	Classe Iniciadora	Classe Participante
FIPA-Request	AchieveREInitiator	AchieveREResponder
FIPA-Query		
FIPA-Propose	ProposeInitiator	ProposeResponder
Versão de Iteração de FIPA-Request FIPA-Query	IteratedAchieveREInitiator	SSIteratedAchieveREResponder
Contract-Net	ContractNetInitiator	ContractNetResponder
FIPA-Subscribe	SubscriptionInitiator	SubscriptionResponder

A seguir, é apresentado como as representações de papel de agente de MAS-ML 2.0 foram implementadas em JAMDER.

4.2.1 Papel de agentes reativos baseados em conhecimento

Estes papéis possuem um atributo que representa as crenças do papel de agente, as quais são incorporadas no agente ao adquirir o papel. A crença do papel de agente também é definida pela classe *Belief*. Caso o agente possua crenças com o mesmo nome definido no papel, seu valor será substituído pelo valor da crença do papel de agente. A classe que representa este papel em JAMDER é *jamder.role.ModelAgentRole* e herda de *AgentRole*, e portanto, as propriedades comuns do papel de agente. Este papel é apropriado apenas para os agentes reativos baseados em conhecimento, ou seja, apenas instâncias de agente do tipo *ModelAgent* podem exercer este tipo de papel.

4.2.2 Papel de agentes proativos

Este papel se caracteriza por possuir como diferencial em relação aos outros dois papéis definidos anteriormente, a propriedade objetivo, também definida pela classe *Goal*. Adicionalmente, como ele possui todas as propriedades definidas em MAS-ML 2.0, então, apenas os agentes baseados em objetivo com plano, agentes baseados em utilidade e agentes baseados em objetivos com planejamento podem exercer este papel. Como a suborganização tem a capacidade de se comportar como um agente, também pode exercer este papel. A classe que define este papel é a *jamder.role.ProactiveAgentRole* e herda de *ModelAgentRole*.

4.3 Organização

A organização consiste no agrupamento de agentes em um SMA [SILVA, 2004], já que representa partições como comunidades, sociedades ou departamentos, onde os agentes habitam e transitam. As organizações estão inclusas em ambientes e só podem habitar em um ambiente, ou seja, as organizações não se movem entre os ambientes e a mesma organização não pode existir em ambientes diferentes. Por possuir muitas características em comum, a organização é definida em JAMDER através da classe *jamder.Organization*, extensão do agente *MAS_MLAgent*. A Figura 7 mostra o nível hierárquico de *Organization*.

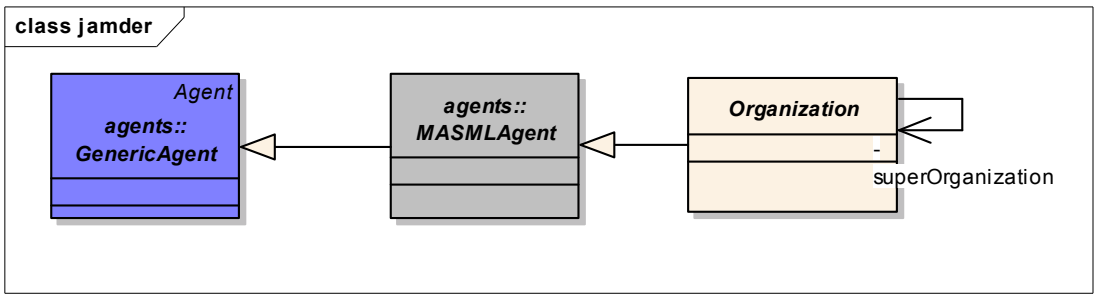


Figura 7 Hierarquia da classe Organization.

A organização também pode conter suborganizações, e estas funcionam ativamente assim como um agente, ou seja, exercem papéis de agente. A lista de suborganizações é definida em *Organization* como um atributo *Hashtable<String, Organization>*, onde *String* é o identificador da suborganização e *Organization* é a instância da suborganização neste atributo. Cada suborganização também pode conter outras suborganizações, pois é uma instância da classe *Organization*.

JADE contém uma classe chamada *ContainerID* que representa o identificador da organização, porém não contempla todas as propriedades definidas em MAS-ML 2.0 para esta entidade. Cada instância de *Organization* possui o atributo *containerID* com o respectivo *container* de JADE associado à organização de JAMDER. Naturalmente, como *Organization* herda de *MAS_MLAgent*, a herança de *ContainerID* não é possível .

Resumidamente, Silva [2004] define as propriedades e relacionamentos da organização da seguinte forma:

- Uma organização possui crenças, objetivos, planos, ações e axiomas;
- Uma organização possui uma identificação;
- As organizações interagem enviando e recebendo mensagens;
- As organizações residem em apenas um ambiente;
- As organizações definem os papéis do agente e os papéis de objeto;
- As organizações definem suborganizações;
- As suborganizações exercem pelo menos um papel em uma organização, neste caso, o papel de agente proativo;
- As suborganizações podem exercer diferentes papéis nas organizações.

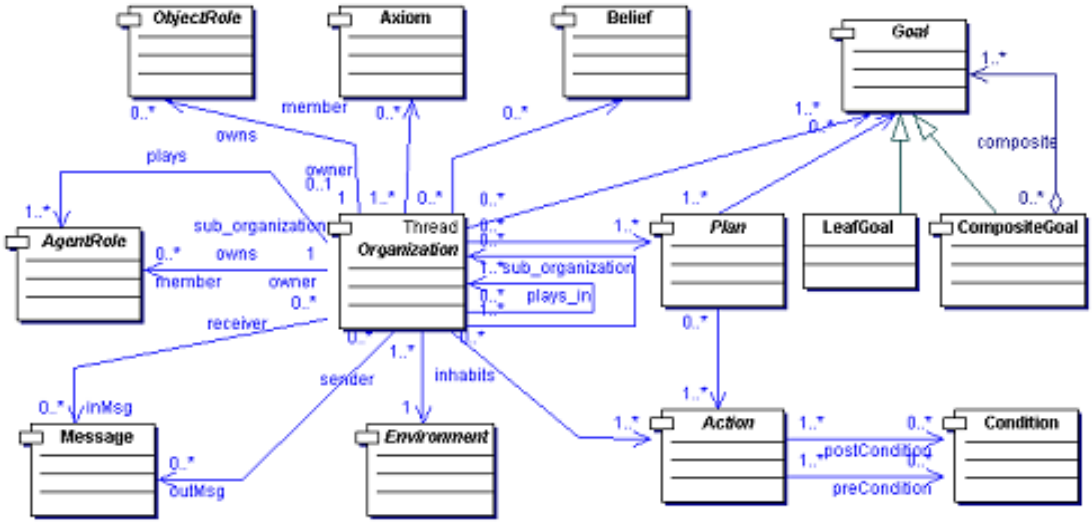


Figura 8 *Organization* e suas classes relacionadas em MAS-ML [SILVA, 2004]

Nas próximas subseções serão apresentadas as características estruturais e comportamentais das organizações. A Figura 8 mostra o diagrama de classe relacionado à organização.

4.3.1 Características Estruturais

Considerando que a classe *Organization*, em JAMDER, herda de *MAS_MLAgent*, consequentemente ela incorpora como características estruturais, os atributos e os seus métodos de acesso, a saber: *Beliefs*, *Goals*, *Plans* e Actions. Além das características identificadas nos agentes, a organização possui os axiomas, os quais podem ser uma regra, lei ou princípio estabelecido, [SILVA, 2004]. Estes axiomas restringem as ações dos agentes e suborganizações que habitam na organização. O conceito de axioma é representado pela classe *jamder.Structural.Axiom*, incluindo três atributos: tipo, nome e valor. Este conjunto de atributos de *Axiom* é idêntico ao especificado para *Property*. Portanto, *Axiom* pode ser definida como subclasse de *Property*. Os axiomas são propriedades particulares das organizações, por conseguinte, seus métodos de acesso *são protected*:

- *getAxiom(String name)* – retorna a instância de um elemento da lista a partir do nome deste elemento;

- *addAxiom(name, Axiom)* – adiciona o elemento *Axiom* identificado pelo nome do elemento;
- *removeAxiom(name)* – remove o elemento da lista a partir do nome do elemento e retorna a lista;
- *removeAllAxiom()* – remove todos os elementos da lista;
- *getAllAxiom()* – retorna a lista de elementos *Axiom*;

Uma instância de suborganização é uma organização, cujo vínculo é estabelecido pelo atributo *superOrganization* do tipo *Organization*.

Apesar da organização representar um compartimento de agentes, estes não são armazenados na forma de atributo. Em vez disso, os agentes são obtidos através dos papéis que a organização possui, pois o papel de agente conhece a organização que participa e o agente que o executa.

4.3.2 Características Comportamentais

Estas características são compostas por ações e planos. Devido os agentes possuírem também, a organização herda estas características, assim como seus métodos de acesso.

A lista de papéis de agente em *Organization* pode funcionar de duas formas:

- No caso de organização, a lista consiste em todos os papéis de agentes aceitos pela organização, ou seja, os papéis armazenados para serem exercidos ou por um agente ou por uma suborganização;
- No caso a instância tenha a função de suborganização, a lista de papéis de agente consiste nos papéis de agente que a suborganização exerce. Para ela executar como suborganização, o atributo *superOrganization* deve conter a instância da classe *Organization* da qual faz parte.

A organização também define os papéis de agente e papéis de objeto. O atributo é especificado como uma instância *Hashtable*<*String, XXX*>, onde XXX pode ser *AgentRole* ou *ObjectRole*. Este atributo armazena todos os papéis de agente ou papel de objeto que o agente ou objeto podem exercer. Seus métodos de acesso seguem a mesma linha dos outros atributos. Na organização, as

propriedades de enviar e receber mensagens são tratadas da mesma forma que no caso dos agentes.

4.4 Objeto e Papel de Objeto

O objeto não pode modificar e nem tem controle sobre seu comportamento [JENNINGS, 2000], em outras palavras, o objeto somente responde às solicitações que lhe foram impostas por uma outra entidade. Ele pode ser representado através da própra classe *Object* de Java.

Analogamente, o papel de objeto guia e restringe o comportamento de um objeto [SILVA, 2004], e apresenta a visão que as outras entidades têm do objeto que exerce este papel. Assim como o papel de agente, o papel de objeto também é definido na organização. A classe *jamder.roles.ObjectRole* foi criada para representar esta entidade e possui as seguintes propriedades, conforme a Figura 9:

- *name* – identificador do papel de objeto;
- *object* – objeto que irá exercer o papel de objeto;
- *owner* – organização da qual o papel de objeto é membro;

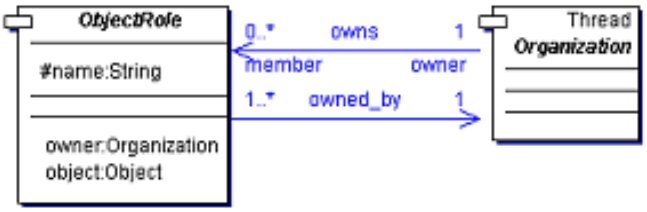


Figura 9 Classe ObjectRole e seus relacionamentos [SILVA, 2004]

4.5 Ambiente

O ambiente é caracterizado por conter as entidades do SMA. Segundo Silva [2004], a classe de ambiente define os métodos, relacionamentos e os atributos iniciais que ligam a classe ambiente às outras classes. Durante seu ciclo de vida, seus atributos ou estados informam as características do ambiente e podem ser modificados através das ações dos agentes. Desta forma, o ambiente representa o *habitat* dos agentes, organizações e objetos. Ainda de acordo com Silva [2004], o ambiente possui as seguintes propriedades:

- Um ambiente possui um identificador;

- Um ambiente é definido por meio de seus atributos e métodos;
- Em um ambiente residem agentes, organizações e objetos;

Em JADE o ambiente é representado pela plataforma onde residem os *containers* (representação de organização em JADE) e os agentes. Quando a plataforma é iniciada, ela está apta a fornecer os serviços necessários para alimentar o ciclo de vida das outras entidades que fazem parte dela, como por exemplo o serviço de busca dos agentes. A classe em JAMDER que define o ambiente é a *jamder.Environment*, porém não herda de nenhuma classe de JADE. Visto que o ciclo de vida da plataforma em JADE possui características internas à mesma para seu funcionamento, em outras palavras, ao iniciar a ferramenta JADE, a plataforma dá início a seus serviços para seu funcionamento. Então, *Environment* atuará como um adaptador à plataforma de JADE.

Cada plataforma iniciada adquire naturalmente um identificador que é uma instância da classe *jade.core.PlatformID.* Similarmente, a classe *Environment* possui o atributo ID, onde é uma instância de *PlatformID*. Adicionalmente, outros atributos que identificam ou localizam a plataforma fazem parte da sua estrutura, tais como: nome e endereço, entre outros.

Baseado nas informações de Silva [2004], o diagrama de classe pertinente à classe *Environment* mostra os relacionamentos que esta possui com as entidades do sistema, como mostrado na Figura 10. Os atributos que definem agentes, organizações e objetos desta classe são instâncias de *Hashtable<String, XXX>*, onde XXX pode ser objetos da classe *Object* ou de classes de JAMDER do tipo *GenericAgent* ou *Organization*.

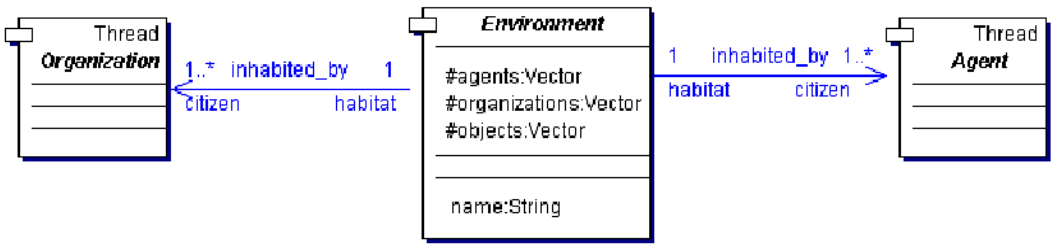


Figura 10 Classe *Environment* e seus relacionamentos [SILVA, 2004]

Ao criar uma instância no ambiente, o construtor *Environment(name, host, port)* recebe o nome da plataforma e o nome e a porta da máquina na qual ficará

armazenada a plataforma. Ao criar um ambiente, a plataforma cria um *container* principal na plataforma que conterá os serviços da plataforma como *Directory Facilitator* (páginas amarelas) e *Agent Management System*, onde este último gerencia as próximas organizações (*containers*) e agentes que serão criados na plataforma.

Além destas propriedades a classe *Environment* também possui outros métodos que vão gerenciar o ambiente, como por exemplo, ao adicionar uma organização no ambiente, o método *addOrganization(String, Organization)* além de incrementar uma instância de *Organization* na lista de organizações, cria um *container* na plataforma JADE.

O processo inverso, ou seja, de remoção das organizações e agentes precisa ser analisado com bastante cautela, pois várias dependências precisam ser levadas em consideração. Antes de uma organização ser excluída, é preciso se desfazer dos relacionamentos com os agentes, suborganizações e objetos que habitam nela. Caso uma organização tenha agentes e objetos, mas se algum agente participa também de outra organização, apenas este cancelará os papéis vinculados à organização que está sendo excluída. Caso contrário, o próprio agente também será excluído. O método *removeOrganization(String key)* implementa este comportamento que, caso a organização contenha suborganizações, será executado de forma recursiva. A Figura 11 mostra o trecho do código de *removeOrganization(String key)*.

```
public class Environment {
  ...
  public void removeOrganization(String key) {
    try {
      Organization org = organizations.get(key);

      // Delete all subOrganizations
      if (!org.getAllSubOrganizations().isEmpty() && org.getAllSubOrganizations().size() >
        0) {
        Enumeration<String> subOrgs = org.getAllSubOrganizations().keys();
        while (subOrgs.hasMoreElements()) {
          String subOrg = (String) subOrgs.nextElement();
          removeOrganization(subOrg);
        }
      }

      Enumeration<String> orgRoleNames = org.getAllAgentRoles().keys();
      AgentRole orgRole = null;
      while (orgRoleNames.hasMoreElements()) {
        String orgRoleName = (String) orgRoleNames.nextElement();
        orgRole = org.getAgentRole(orgRoleName);
        GenericAgent agent = orgRole.getPlayer();

        Enumeration<String> aRoleNames = agent.getAllAgentRoles().keys();
        AgentRole aRole = null;
```

```
        boolean hasOtherOrgs = false;

        // Cancel the role with agents
        while (aRoleNames.hasMoreElements()) {
          String aRoleName = (String) aRoleNames.nextElement();
          aRole = agent.getAgentRole(aRoleName);
          if (aRole.getOwner() != org) {
            aRole.getOwner().getContainerController().acceptNewAgent(agent.getName(),
       agent);
            hasOtherOrgs = true;
            break;
          }
        }

        // When agent have not other organizations, it will be deleted
        if (!hasOtherOrgs) {
          removeAgent(agent.getName());
        }

        // Removes the instance of AgentRole
        org.removeAgentRole(orgRoleName);
      }

      // Delete the container on JADE and JAMDER
      org.getContainerController().kill();
      organizations.remove(key);
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
  ...
}
```

Figura 11 Trecho do método removeOrganization de Environment

Partindo da premissa de que o ambiente conhece todos os agentes que suporta, o procedimento de criação dos agentes se dá nesta classe através do método *addAgent(String key, GenericAgent agent)*, onde *key* é o nome do agente e *agent* é a própria instância do mesmo. A remoção apenas do agente é mais simples do que a da organização, visto que possui menos dependências. O processo de remoção do agente consiste, inicialmente, na verificação dos papéis que ele exerce para ver em quais organizações o agente está lotado. Ao remover o contrato com o papel de agente, a instância deste passa a não existir. O método *removeAgent(String name)* realiza este comportamento.

As mudanças que o agente percebe acontecem no próprio ambiente. Estas mudanças são oriundas das ações que os próprios agentes executam, ou seja, quando um agente executa alguma ação, ele pode modificar algum estado do ambiente.

4.6 Movimentação de Agentes

Segundo Silva [2004], um agente pode mudar de um ambiente para outro a fim de encontrar um serviço e alcançar seus objetivos com base no relacionamento

entre ambientes. Para Masif [2004], um ambiente pode permitir ou negar que um agente entre ou saia dele.

Mesmo que o agente possa se movimentar entre ambientes, não é possível estar em dois ambientes ao mesmo tempo. Consequentemente, o agente precisa cancelar todos os papéis e relacionamentos que exerce antes de se locomover para outro ambiente, assim como nas organizações em que ele exerce papel. Ao entrar no novo ambiente, o agente é instanciado em uma organização e passa a exercer algum papel de agente desta organização. Se a migração do agente acontecer entre organizações, este muda o status dos papéis de agente para CHANGE e suas ações são finalizadas.

Em JADE, todo agente dispõe de três métodos que o auxiliam no momento em que este estiver se locomovendo. Estes métodos são sobrecarregados em *GenericAgent* para adaptar a migração dos agentes a fim de atualizar as organizações e o ambiente. O método *beforeMove()* contém o mecanismo responsável por executar uma ação antes do agente se locomover. Em seu código, o status dos papéis de agente é modificado para CHANGE. O método *afterMove()* é executado depois da migração do agente e, em seu conteúdo, o desenvolvedor pode fornecer a informação do que pode ser feito depois que o agente se locomover. Se a locomoção for entre organizações, os papéis de agente da organização de destino que estejam no status CHANGE mudam para o status ACTIVATE e as ações atribuídas a este papel voltam a ser exercidas. Se a locomoção for entre ambientes, ele irá exercer as ações do papel de agente que for obtido na organização do ambiente de destino. Por fim, o método *doMove(Location)*, tem a funcionalidade de verificar o próprio comportamento da migração. O parâmetro *location* é uma superclasse de *ContainerID* e *PlatformID*, e é responsável por informar o *ID* da organização ou do ambiente, respectivamente. Caso a locomoção seja entre organizações, é preciso verificar se a organização de destino tem algum papel que o agente exerce. Se sim, o agente muda de organização, caso contrário, é gerada uma mensagem de alerta. Se a locomoção for entre ambientes, o agente é transferido para o novo ambiente designado.

5 GERAÇÃO DE CÓDIGO

Para a abordagem da geração de código orientada a modelos foi adotada a ferramenta *MAS-ML Tool* para criar diagramas e posteriormente serem representadas em código JAMDER. MAS-ML *Tool* foi desenvolvido como um *plugin* para o Eclipse [ECLIPSE, 2011], onde os usuários podem desenvolver a modelagem de SMAs com MAS-ML 2.0, ao mesmo tempo em que utilizam os recursos oferecidos pela plataforma Eclipse. O uso desta plataforma facilita a geração de código, visto que os *plugins* integrados ao ambiente de desenvolvimento podem trabalhar em conjunto em uma mesma ferramenta. Consequentemente, outro *plugin* do Eclipse, específico para geração de código, foi incorporado à plataforma para a geração de código, a saber, o Acceleo.

Então, de acordo com a arquitetura MDA, estas ferramentas podem ser aplicadas nas seguintes etapas da geração de código baseado em modelos:

- PIM: MAS-ML *Tool*;
- PSM: Java (*framework* JAMDER):
- PDM: *templates* Acceleo;

5.1 Ferramentas utilizadas

As ferramentas de modelagem e de geração de código utilizadas nesta dissertação são detalhadas a seguir.

5.1.1 MAS-ML *Tool*

MAS-ML Tool, [FARIAS et al., 2009], [GONÇALVES et al., 2010a] e [GONÇALVES et al., 2010b], segue uma abordagem dirigida por modelos, sendo que o modelo central é o metamodelo da linguagem MAS-ML 2.0. Segundo Gonçalves [2009], o objetivo desta ferramenta é fornecer um ambiente de modelagem no qual os desenvolvedores possam trabalhar com os conceitos do domínio do problema, ao mesmo tempo em que utilizam explicitamente conceitos definidos no domínio da solução, neste caso, os conceitos e abstrações definidos no paradigma de SMAs.

A Figura 12 mostra a ferramenta MAS-ML *Tool* em funcionamento juntamente com alguns de seus componentes principais [FARIAS et al., 2009]:

(A) *Package Explorer* - Tem como funcionalidade central permitir a organização dos arquivos em uma estrutura de árvore a fim de que seja possível um melhor gerenciamento e manipulação dos arquivos;

(B) *Modeling View* - Os modelos que são criados precisam necessariamente ser visualizados com o objetivo de atender dois requisitos básicos quando se usa modelos: compreensibilidade e a comunicação. Diante dessa necessidade, a *modeling view* permite os desenvolvedores visualizar e editar os modelos de forma interativa;

(C) *Nodes Palette* - Os construtores que podem fazer parte dos diagramas encontram-se na *nodes palette*. Sendo assim, os desenvolvedores podem criar instâncias desses construtores, as quais são visualizadas na *modeling view*.

(D) *Relationship Palette* - Os relacionamentos que podem ser estabelecidos entre os construtores presentes na *nodes palette* são disponibilizados na *relationship palette*.

(E) *Properties View* - Permite manipular de forma precisa as propriedades dos modelos. Exibe as propriedades definidas no metamodelo da MAS-ML quando o modelo é exibido e selecionado na *modeling view*.

(F) *Problems View* - Caso exista alguma inconsistência no modelo, ela será mostrada em *problems view*. Esta funcionalidade é particularmente importante para viabilizar o uso de modelagem de SMAs dentro do contexto do desenvolvimento dirigido por modelos, na qual modelos são vistos como artefatos de primeira ordem.

(G) *Outline View* - Uma vez que um modelo tenha sido criado, um overview da distribuição dos elementos presentes no mesmo poderá ser visualizado através da *outline view*.

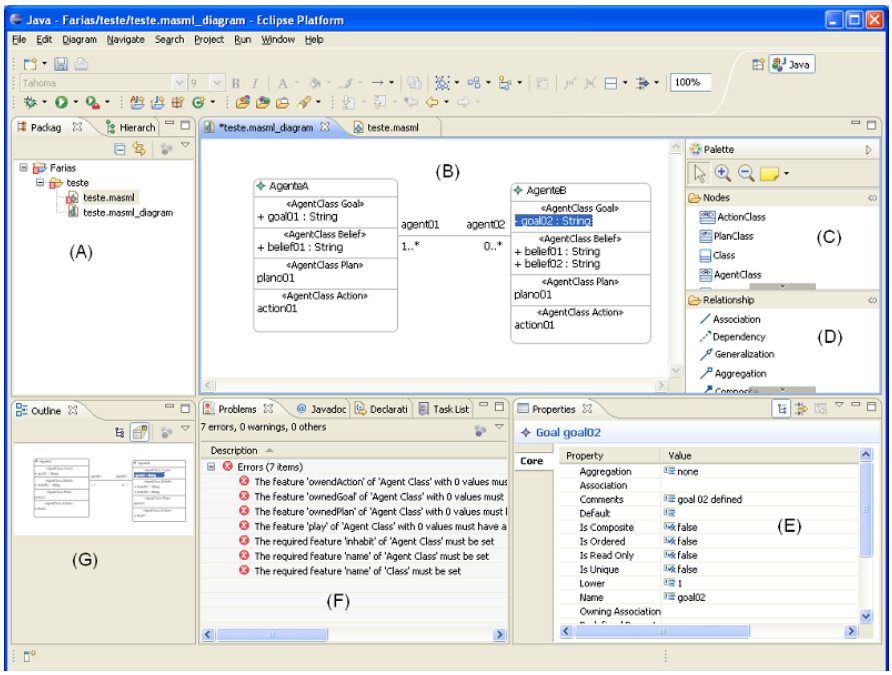


Figura 12 Visão geral da ferramenta MAS-ML Tool

MAS-ML *Tool* utiliza recursos de EMF (*Eclipse Modeling Framework*), GEF (*Graphical Editing Framework*) e GMF (*Graphical Modeling Framework*) que facilita a construção de ferramentas baseadas em modelagem. Cada diagrama gerado na ferramenta *MAS-ML Tool*, produz dois arquivos, os quais são:

- .masml_diagram – armazena a estrutura visual do diagrama, ou seja, os aspectos de aparência de cada entidade como cor, tamanhos e outros;
- .masml – armazena os dados das entidades, como crenças, objetivos, funções e outros aspectos estruturais e comportamentais definidos em MAS_ML 2.0;

Como os arquivos gerados são baseados em EMF, a estrutura dos mesmos é gerada no formato XMI (*XML Metadata Interchange*), similar a XML. XMI é um dos padrões definidos pela OMG (*Object Management Group*) e seu uso é utilizado nas ferramentas de modelagem para armazenamento dos metadados. A Figura 13 mostra um exemplo da estrutura XMI de um arquivo .masml.

```
<?xml version="1.0" encoding="UTF-8"?>
<masml:MasmlClassDiagram xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:masml="masml">
  <ownedObjectRoleClass name="SalmitoT"/>
```

```
  <ownedAgentRoleClass name="Teste">
    <ownedGoal xsi:type="masml:Goal"/>
    <ownedDuty xsi:type="masml:Duty" name="Duty:String"/>
    <ownedRight xsi:type="masml:Right" name="Right:Object"/>
    <ownedBelief name="BeliefT" default="ator" aggregation="composite"/>
  </ownedAgentRoleClass>
</masml:MasmlClassDiagram>
```

Figura 13 Exemplo de XMI para o arquivo .masml

MAS-ML *Tool* disponibiliza três tipos de diagramas: diagrama de papel, diagrama de organização e de classes. Cada um dos diagramas gera um arquivo de modelagem, porém os mesmos não são interligados. Esta ainda é uma limitação da ferramenta MAS-ML *Tool* na versão atual, pois nem todos os relacionamentos podem ser modelados em uma única modelagem (único arquivo XMI).

A partir das informações fornecidas pela ferramenta MAS-ML *Tool* através dos arquivos originados (.masml), é possível aplicá-los para geração de código através de outro *plugin*, o Acceleo.

5.1.2 Acceleo

O *plugin* do eclipse utilizado para este processo é o Acceleo, cujo uso vem sendo bastante utilizado por parte da comunidade acadêmica [SOUZA, 2011] para geração de código a partir de modelos envolvendo o Eclipse, [ACCELEO, 2011]. Ele permite que o código seja gerado repetidamente e pode trabalhar com algum modelo definido pelo usuário que esteja de acordo com o EMF, neste caso, MAS-ML *Tool*.

O Acceleo [ACCELEO, 2011] é resultado da experiência de vários arquitetos que tem praticado trabalhos baseados em MDA. Ele possui diferentes módulos que representam as estruturas de todas as possíveis classes ou artefatos para transformação em uma plataforma específica, por exemplo, C#, Java, PHP, entre outros. A Figura 14 mostra este processo, onde a geração de código começa pelo modelo definido através da plataforma Eclipse e logo após, um módulo é utilizado, dependendo da linguagem de implementação a ser utilizada na geração. Ao executar um módulo, ele verifica a estrutura do modelo e a partir destes passos o código pode ser gerado.

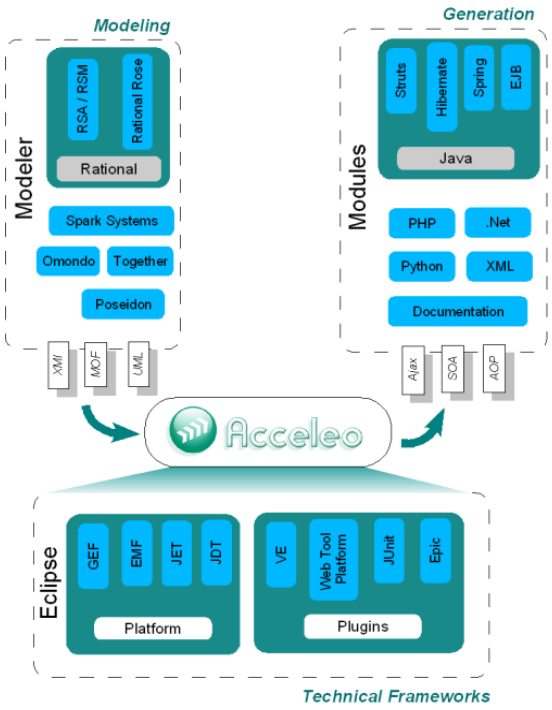


Figura 14 Processo de geração Acceleo [ACCELEO, 2011].

Um módulo é composto de vários *templates* que descrevem os parâmetros necessários para gerar o código fonte a partir de um modelo. Os *templates* geram o texto da forma ou sequência da forma que foi desenvolvido extraindo dados de um modelo abstrato, os arquivos .masml por exemplo. Para formalizar a geração de código a partir de Acceleo, é preciso estabelecer um *template* para cada entidade através de uma linguagem definida pela OMG, o MTL (*Model Transformation Language*) [OMG, 2011]. O Acceleo utiliza estes *templates* que ficam armazenados em arquivos .mtl, ou seja, cada arquivo .mtl pode conter vários *templates* necessários para a geração.

Ao iniciar um projeto em Acceleo, algumas informações devem informadas para o projeto reconhecer a estrutura do metamodelo (ou módulo) MAS-ML 2.0. Ele necessitará saber: (i) a pasta principal, (ii) um identificador do projeto (generate), (iii) o metamodelo a ser utilizado, no caso o masml e (iv) uma classe principal, neste caso, class. Após a criação do projeto, a classe *generate* é criada automaticamente, sendo utilizada para leitura dos diagramas de MAS-ML *Tool*. O módulo utilizado no arquivo .mtl é chamado masml, pois na ferramenta MAS-ML *Tool* foi definido este identificador para informar quais classes ou artefatos em MAS-ML *Tool* podem ser

utilizados. A Figura 15 mostra as informações necessárias para a criação do projeto Acceleo.

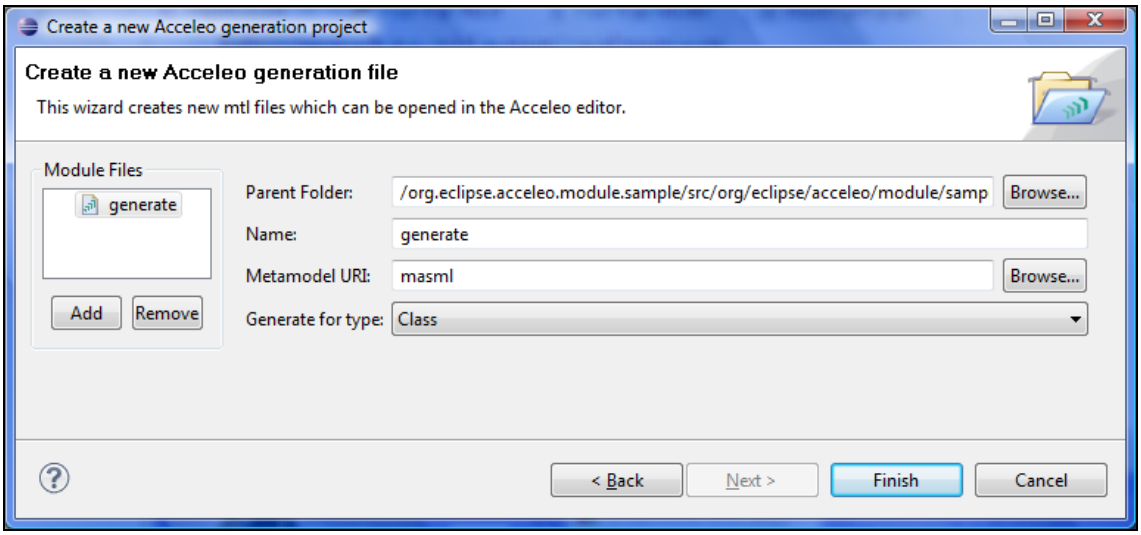


Figura 15 Tela de criação do projeto em Acceleo no Eclipse.

Além da classe *generate*, o projeto também cria o arquivo *generate.mtl*, o qual conterá os templates necessários. A classe *generate* não precisa ser alterada e o exemplo do arquivo .mtl é mostrada na Figura 16.

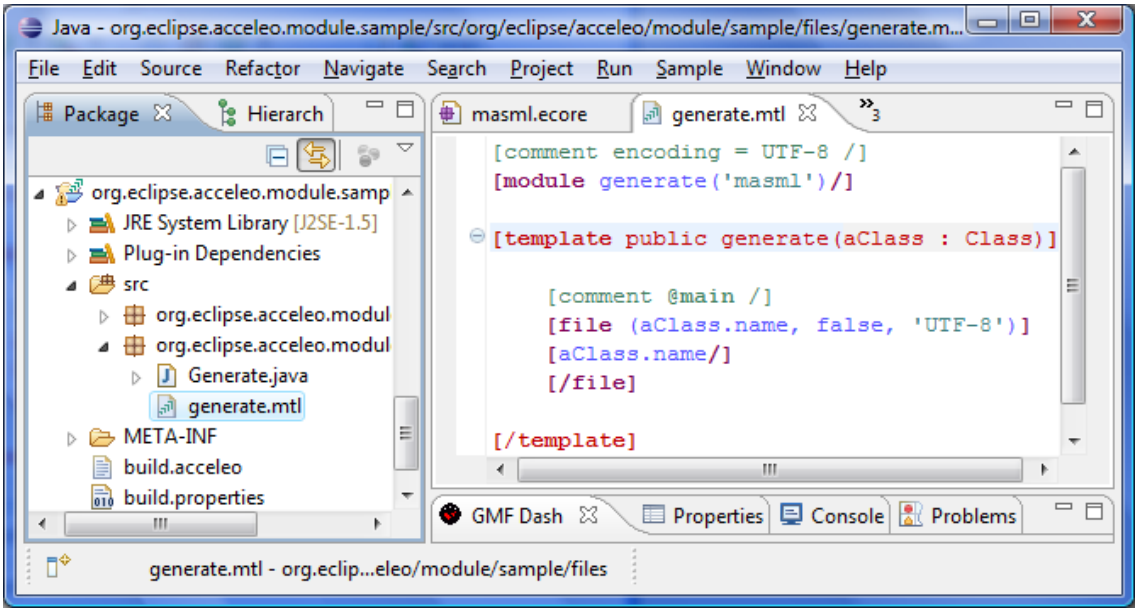


Figura 16 Associação do módulo masml para ser lido por generateJava.

Por fim, ao executar o arquivo .mtl contendo o código dos templates para as entidades ou classes necessárias na geração, ele irá solicitar:

1. O projeto criado no Eclipse;
2. O nome da classe que lerá o arquivo .mtl. Ex.: Generate;
3. Um arquivo de modelagem. Ex.: .moodle.masml;
4. A pasta ou pacote que os arquivos gerados ficarão armazenados;

A Figura 17 exibe a tela de execução do *template* Acceleo.

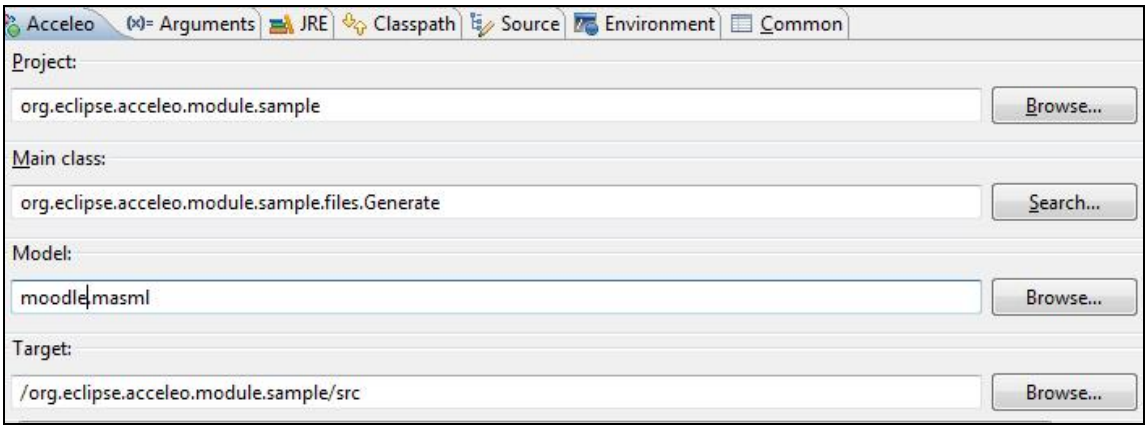


Figura 17 Tela de execução do template Acceleo.

5.2 Templates Acceleo para MAS-ML 2.0

Uma vez definidos a ferramenta de modelagem, MAS-ML *Tool*, e o *framework* que será utilizado, JAMDER (Java), pode-se dar início à criação do mapeamento das entidades e/ou classes necessárias nos *templates* de Acceleo. As classes geradas necessitam somente herdar de uma das classes de JAMDER e acrescentar instâncias das características estruturais e comportamentais, tais como *Belief*, *Goal*, entre outros.

Os relacionamentos definidos em MAS-ML 2.0 estão configurados implicitamente na geração de código a partir do diagrama projetado em MAS-ML *Tool*, visto que eles são utilizados para fazer associações ou com as outras entidades existentes. Todas as validações destes relacionamentos também já estão contempladas pela ferramenta.

O fato da ferramenta MAS-ML *Tool* disponibilizar cada diagrama modelado em um arquivo XMI diferente acarreta na geração de código ocorrendo apenas com os relacionamentos do diagrama presente no XMI de entrada. O Acceleo pode atender aos relacionamentos propostos em MAS-ML 2.0, pois os *templates* tratam dos

relacionamentos propostos e verifica se o mesmo está presente no modelo, caso o relacionamento não exista no diagrama modelado em MAS-ML *Tool*, o *template* Acceleo verifica as próximas propriedades que o modelo contém.

O ambiente tem uma função primordial de armazenamento das outras entidades do SMA. Em seguida, as próximas entidades no processo de criação são os objetos e as organizações. A próxima etapa da criação de entidades envolve os agentes e suborganizações, onde cada um deles possui uma instância de papel de agente inicial, criada em conjunto e associadas a estas entidades. Em relação ao papel de objeto, este também é criado na terceira etapa e contém em sua associação, o objeto. A Figura 18 mostra este processo de criação.

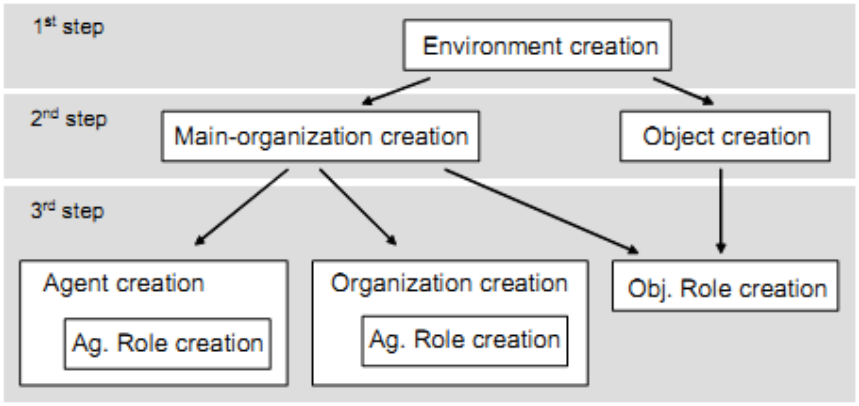


Figura 18 Processo de criação das entidades. [SILVA, 2004]

A seguir, têm-se os *templates* propostos para a geração de código em JAMDER a partir das entidades de MAS-ML 2.0.

5.2.1 Ambiente

O ambiente é uma instância de uma classe que herda da classe *Environment* de JAMDER e sua representação ocorre através de *EnvironmentClass* de MAS-ML *Tool*. Como esta instância herda os métodos definidos em *Environment*, é necessário apenas que na geração do código o construtor desta classe chame a superclasse e crie as organizações, agentes e papéis de agente, nesta ordem. A necessidade da criação de instâncias de papel de agente é apenas para vincular as instâncias de organização e agente, onde o construtor do papel de agente já faz este vínculo. O mapeamento da criação de instâncias de organização e agentes acontece através do relacionamento *inhabit* entre o ambiente e estas entidades. Os objetos e

papéis de objeto também são criados no ambiente, onde os papéis são obtidos pela organização através do relacionamento *ownership*. Por sua vez, os objetos são obtidos pelo papel de objeto através do relacionamento *play*.

Caso sejam necessários métodos ou atributos adicionais, a ferramenta MASML *Tool* disponibiliza os componentes *Operation* e *Property* para esta necessidade, respectivamente. A Figura 19 mostra o *template* para gerar o ambiente.

```
[template public generateJava(c : EnvironmentClass)]
[comment @main /]
[file (c.name + '.java', false, 'UTF-8')]
import jamder.environment;
import jamder.Organization;
import jamder.roles.AgentRole;
import jamder.agents.GenericAgent;
import jamder.behavioural.*;
public class [c.name /] extends Environment {
  public [c.name /] (String name, String host, String port) {
        super(name, host, port);
    [for(i : Inhabit | c.inhabit)]
       [if (i.org -> size() > 0) ]
      Organization [i.org.name/] = new Organization("[i.org.name/]", this, null);
          addOrganization("[i.org.name/]", [i.org.name/]);
          [if (i.org.play -> size() > 0) ]
          [let ar : AgentRoleClass = i.org.play.agentRole]
            AgentRole [ar.name/] = new AgentRole("[ar.name/]", [ar.ownership.owner.name/],
[i.org.name/]);
          [/let]  [/if]

      [for(ow : Ownership | i.org.ownership)]
            [for(ob : ObjectRoleClass | ow.objectRole)]
              Object [ob.play.name/] = new Object();
              addObject("[ob.name/]", [ob.name/]);
              ObjectRole [ob.name/] = new ObjectRole("[ob.name/]", [i.org.name/],
[ob.play.name/]);
            [/for]
      [/for]
       [/if]
    [/for]

    [for(i : Inhabit | c.inhabit)]
       [if (i.agentClass -> size() > 0) ]
          [let a : AgentClass = i.agentClass]
      GenericAgent [a.name /] = new [a.name /]("[a.name /]", this, null);
          AgentRole [a.play.agentRole.name/] = new AgentRole("[a.play.agentRole.name/]",
[a.play.agentRole.ownership.owner.name/], [a.name/]);
          addAgent("[a.name /]", [a.name /]);
          [/let]
       [/if]
    [/for]
  }
  // Additional attributes
  [for(p : Property | c.ownedProperty)]
     [p.visibility/] [p.type.toString()/] [p.name/];
  [/for]

  // Additional methods
  [for(o : Operation | c.ownedOperation)]
     [o.visibility /] [o.returnValue /] [o.name /]
        ([for(p:Parameter | o.parameter) separator(',')] [p.type/] [p.name/] [/for]) {}
  [/for]
}
[/file]
[/template]
```

Figura 19 Template para a geração do Ambiente

Uma característica importante a observar é que o papel de agente necessita saber qual o agente ou suborganização está exercendo este papel, ao mesmo tempo em que o agente ou suborganização necessita saber qual seu papel de agente inicial. Para resolver este problema de ciclo, na criação do Agente ou suborganização, o parâmetro de papel de agente inicia como nulo. Na criação do papel de agente, seu construtor recebe como parâmetro a instância do agente ou suborganização, onde este construtor configura no agente ou suborganização, o papel que irá exercer inicialmente através do método addAgentRole(AgentRole).

5.2.2 Papel de Objeto e Objeto

O papel de objeto em JAMDER é representado pela entidade *ObjectRoleClass* de MAS-ML *Tool*. A estrutura desta classe não possui muitos atributos, por sua vez, a geração de novos papéis de objeto apenas herda desta classe. Enquanto que a entidade Objeto é qualquer objeto de Java e isto o Acceleo já cria através de sua entidade pré-definida *Class*. A Figura 20 mostra os detalhes deste *template*.

```
[template public generateJava(c : ObjectRoleClass)]
[comment @main /]
[file (c.name + '.java', false, 'UTF-8')]
import jamder.roles.ObjectRole;
import jamder.Organization;
public class [c.name /] extends ObjectRole {
   //Constructor
   public [c.name /] (String name, Organization owner, Object object) {
          super(name, owner, object);
   }
}
[/file]
[/template]
```

Figura 20 Template para ObjectRole.

5.2.3 Papel de Agente

A entidade *AgentRoleClass* de MAS-ML *Tool* oferece suporte para os três tipos de papel de agente definidos por MAS-ML 2.0. A diferença é estabelecida pela presença dos componentes que o papel possui. Se o papel não tiver crenças e objetivos, o papel de agente será do tipo *AgentRole* de JAMDER. Se o papel não tiver apenas crenças, o papel de agente será do tipo *ModelAgentRole* de JAMDER. E se o papel contiver toda a estrutura disponível, ele será do tipo *ProactiveAgentRole* de JAMDER. O *template* em .mtl que trata de papel de agente implementa esta verificação e cria a herança correspondente às características encontradas. Ao final deste *template*, é realizada a chamada do método *initialize()*

de *AgentRole*, *framework* JAMDER, para checar os direitos e deveres que serão exercidos pelo papel. Em relação ao protocolo, como em JADE existem diferentes tipos e eles, de alguma forma, herdam de *Behaviour*, os protocolos presentes no papel contêm apenas o nome do protocolo e um objeto nulo, onde o desenvolvedor precisa informar qual tipo de protocolo existente em JADE será usado. Este *template* pode ser visto na Figura 21.

```
[template public generateJava(c : AgentRoleClass)]
[comment @main /]
[file (c.name + '.java', false, 'UTF-8')]
import jamder.structural.*;
import jade.core.behaviours.Behaviour;
import jamder.Organization;
import jamder.agents.GenericAgent;
import jamder.roles.*;
import jamder.behavioural.*;

[if (c.superClass->size() > 0) ]
public class [c.name /] extends [c.superClass.name /] {
[else]
        [if ((c.ownedBelief->size() <= 0) and (c.ownedGoal->size() <= 0)) ]
public class [c.name /] extends AgentRole {
        [elseif ((c.ownedBelief->size() > 0) and (c.ownedGoal->size() <= 0)) ]
public class [c.name /] extends ModelAgentRole {
        [elseif ((c.ownedBelief->size() > 0) and (c.ownedGoal->size() > 0))]
public class [c.name /] extends ProactiveAgentRole {
        [/if]
[/if]
   //Constructor
   public [c.name /] (String name, Organization owner, GenericAgent player) {
   super(name, owner, player);

   [for(b : Belief | c.ownedBelief)]
   addBelief("[b.name/]", new Belief("[b.name/]", "[b.type/]", "[b.default/]"));
   [/for]

   [for(g : Goal | c.ownedGoal)]
   addGoal("[g.name/]", new LeafGoal("[g.name/]", "[g.type/]", "[g.default/]"));
   [/for]

   [for(r : Right | c.ownedRight)]
   addRight("[r.name/]", new Right());
   [/for]

   [for(d : Duty | c.ownedDuty)]
   addDuty("[d.name/]", new Duty());
   [/for]

   [for(p : ProtocolClass | c.protocol)]
   addProtocol("[p.name/]", null);
   [/for]

    initialize();
   }
}
[/file]
[/template]
```

Figura 21 Template para AgentRole.

5.2.4 Agente

A geração do agente é análoga à do papel de agente, pois existem cinco tipos de agentes e cada um herda da classe correspondente ao agente em JAMDER,

sendo que, isto dependerá dos componentes que o agente contiver. Seguindo a definição dos agentes em JAMDER, as características existentes em um agente determina o tipo de agente, desta forma, uma das classes agente de JAMDER. Observe que a modelagem do agente necessita estar coerente com seu tipo, ou seja, sua estrutura deve estar de acordo com a especificação MAS-ML 2.0, a fim de em sua geração, as propriedades do agente também condizerem com seu tipo de agente. Para facilitar a definição do tipo de agente ou herança a partir de sua estrutura, é estabelecida a verificação dos componentes do diagrama na seguinte ordem:

- Se na modelagem o agente herda de outro agente, então na geração ele também herdará deste outro agente;
- Se o agente não define crenças (*beliefs*) e objetivos (*goals*), este agente herda de *ReflexAgent*;
- Se define crenças, mas não tiver objetivos, este agente herda de *ModelAgent*;
- Se define um plano pré-definido (*plan*), este agente herda de *MASMLAgent*;
- Se existir um plano a ser definido (*planning*) e não possuir *utilityFunction()*, este agente herda de *GoalAgent*;
- Se existir um plano a ser definido (*planning*) e possuir um *utilityFunction()*, este agente herda de *UtilityAgent*;

O componente *Action* de *AgentClass*, definido na ferramenta MAS-ML *Tool*, funciona de duas formas na representação dos agentes. Cada atributo deste componente possui um campo chamado *ActionSemantics* que pode ou não ser preenchido. O preenchimento do campo *ActionSemantics* pode conter e funcionar em uma das seguintes situações:

- <<*Next-Function*>>: indica que o agente contém a função do tipo *nextFunction()* e é útil para os agentes reativos com conhecimento (*ModelAgent*);
- <<*Formulate-Problem-Function*>>: indica que o agente contém a função do tipo *formulateProblemFunction*() e é usado para os agentes com planejamento (*GoalAgent*);

- <<*Formulate-Goal-Function*>>: indica que o agente contém a função do tipo *formulateGoalFunction()* e é usado para os agentes com planejamento (*GoalAgent*);
- <<*Utility-Function*>>: indica que o agente contém a função do tipo *utilityFunction()* e é usado para os agentes com planejamento (*UtilityAgent*);

Se o campo *ActionSemantics* não for preenchido, isso significa que este atributo é uma das ações que o agente é capaz de executar, ou seja, é uma instância *Action* de JAMDER. Um ponto importante sobre as ações é que, em MAS-ML, elas podem possuir precondições e pos-condições, porém na versão atual de MAS-ML *Tool*, esta característica ainda não é contemplada. Por outro lado, caso o campo *ActionSemantics* seja preenchido, este funciona como algum dos métodos que o agente executa, dependendo da estrutura do agente. A atribuição ou definição dos métodos vai depender do tipo de agente que o projetista quer definir.

Apesar de estes métodos conterem algum estereótipo que identifica sua função, o método também contém um nome, entretanto, em JAMDER, os nomes dos métodos são abstratos e já estão definidos. Para resolver este problema, a geração de código terá dois métodos para cada propriedade *ActionSemantics* preenchida, no caso, o método de JAMDER e o método com o nome definido na modelagem, onde o primeiro faz referência ao segundo método. Para melhor entendimento, se o agente contém a propriedade <<*Next-Function*>>*funcaoProximoGrupos*, o gerador gera o método *nextFunction()* o qual chama o método *funcaoProximoGrupos()*.

Caso o agente contenha algum atributo no compartimento *Planning*, isto indica que este agente (*GoalAgent*) conterá a função *planning()*, usada para montar o plano em tempo de execução.

A criação do plano acontece de duas formas, *Plan* ou *planning*. A primeira forma é um plano predefinido em tempo de modelagem, onde em sua estrutura há o atributo *ownedAction* que guarda as ações do mesmo e são parte do agente. A segunda forma é um plano sem ações, mas que pode ser montado em tempo de execução pelo agente. Os dois tipos de plano de MAS-ML *Tool* são instâncias da classe *Plan* de JAMDER.

As listas de mensagens enviadas e recebidas do agente não precisam ser definidas na criação do agente, visto que ao ser iniciado ele não enviou ou recebeu qualquer mensagem. Os métodos de acesso a estas listas são herdadas da classe *GenericAgent* de JAMDER.

A Figura 22 mostra o *template* de criação para todos os tipos de agente definidos em MAS-ML 2.0.

```
[query public possuiUtility(actions : OrderedSet(ActionClass)) : Boolean =
actions.actionSemantics.toString().equalsIgnoreCase('<<Utility-Function>>') /]

[template public generateJava(c : AgentClass) ]
[comment @main /]
[file (c.name + '.java', false, 'UTF-8')]
import jamder.behavioural.*;
import jamder.Environment;
import jamder.roles.AgentRole;
import jamder.structural.*;
import java.util.List;
import jamder.agents.*;

    [if (c.superClass->size() > 0) ]
public class [c.name /] extends [c.superClass.name /] {
       [elseif ((c.ownedBelief->size() <= 0) and (c.ownedGoal->size() <= 0)) ]
public class [c.name /] extends ReflexAgent {
       [elseif ((c.ownedBelief->size() > 0) and (c.ownedGoal->size() <= 0)) ]
public class [c.name /] extends ModelAgent {
       [elseif (c.ownedPlan->size() > 0) ]
public class [c.name /] extends MASMLAgent {
       [elseif (c.ownedPlanning->size() > 0) ]

       [if (possuiUtility(c.owendAction).toString().contains('true'))]
public class [c.name/] extends UtilityAgent {
       [else]
public class [c.name /] extends GoalAgent {
       [/if]

[/if] [/if]

   //Constructor
   public [c.name.toUpperFirst() /] (String name, Environment env, AgentRole agRole) {
   super(name, env, agRole);

   [for(b : Belief | c.ownedBelief)]
      addBelief("[b.name.concat('B')/]", new Belief("[b.name.concat('B')/]", "[b.type/]",
       "[b.default/]"));
   [/for]

   [for(g : Goal | c.ownedGoal)]
      Goal [g.name.concat('G')/] = new LeafGoal("[g.name.concat('G')/]", "[g.type/]",
       "[g.default/]");
      addGoal("[g.name.concat('G')/]", [g.name.concat('G')/]);
   [/for]

   [for(ac : ActionClass | c.owendAction)]
     [if (ac.actionSemantics.toString().trim().size() <= 0)]
       Action [ac.name.concat('Ac')/] = new Action("[ac.name.concat('Ac')/]", null, null);
       addAction("[ac.name.concat('Ac')/]", [ac.name.concat('Ac')/]);
     [/if]
   [/for]

   [for(p : Perception | c.ownedPerception)]
      [if (c.ownedGoal->size() <= 0)]
        addPerceive("[p.name/]", null);
      [/if]
   [/for]

   [if (c.ownedGoal->size() > 0)]
```

```
      public void percept(String perception) { }
   [/if]

   [for(p : PlanClass | c.ownedPlan)]
     Plan [p.name.concat('Plan')/] = new Plan("[p.name.concat('Plan')/]",
        [p.owendGoal.name.concat('G')/]);
     addPlan("[p.name.concat('Plan')/]", [p.name.concat('Plan')/]);
     [for(ac : ActionClass | p.ownedAction)]
        [p.name.concat('Plan')/].addAction("[ac.name.concat('Ac')/]",
        [ac.name.concat('Ac')/]);
     [/for]
   [/for]

   [for(p : Planning | c.ownedPlanning)]
      Plan [p.name.concat('Plan')/] = new Plan("[p.name.concat('Plan')/]", null);
      addPlan("[p.name.concat('Plan')/]", [p.name.concat('Plan')/]);
   [/for]
   }

[if (c.ownedPlanning->size() > 0)]
protected Plan planning(List<Action> actions){
       return null;
}
[/if]

[if (c.ownedPlan->size() > 0)]
public void percept(String perception) { }
[/if]

[for(a : ActionClass | c.owendAction)]
 [if (a.actionSemantics.toString().equalsIgnoreCase('<<Next-Function>>'))]
protected Belief nextFunction(Belief belief, String perception) {
       return [a.name/]( belief, perception);
}
private Belief [a.name/](Belief belief, String perception) {
       return null;
}
 [elseif (a.actionSemantics.toString().equalsIgnoreCase('<<Formulate-Problem-Function>>'))]
protected List<Action> formulateProblemFunction(Belief belief, Goal goal) {
       return [a.name/]( belief, goal);
}
private List<Action> [a.name/](Belief belief, Goal goal) {
       return null;
}
 [elseif (a.actionSemantics.toString().equalsIgnoreCase('<<Formulate-Goal-Function>>'))]
protected Goal formulateGoalFunction(Belief belief) {
       return [a.name/]( belief);
}
private Goal [a.name/](Belief belief) {
       return null;
}
 [elseif (a.actionSemantics.toString().equalsIgnoreCase('<<Utility-Function>>'))]
protected Integer utilityFunction(Action action) {
       return [a.name/]( action);
}
private Integer [a.name/](Action action) {
       return 0;
}
 [/if]
[/for]
}
[/file]
[/template]
```

Figura 22 Template para *Agent*.

5.2.5 Organização

Similar à estrutura dos agentes, a organização dispõe de vários componentes em sua estrutura. A definição de crenças, objetivos, ações e planos é idêntica à criação destes componentes nos agentes, porém, a diferença acontece em relação a

outro componente que a organização possui, os axiomas. A organização é representada pela entidade *OrganizationClass* da ferramenta MAS-ML *Tool* e assim como as outras entidades, ele herda de uma classe de JAMDER, no caso, *Organization*.

O processo de criação das suborganizações é o mesmo da organização, onde a exceção se faz no parâmetro *org* que o construtor possui. Quando este parâmetro não for nulo, ou seja, possuir alguma instância de *Organization*, isso significa que a organização corrente possui uma superorganização. Esta regra é mais detalhada em JAMDER. A Figura 23 mostra o *template* para *Organization*.

```
[template public generateJava(c : OrganizationClass)]
[comment @main /]
[file (c.name + '.java', false, 'UTF-8')]
import jamder.Organization;
import jamder.roles.*;
import jamder.structural.*;
import jamder.behavioural.*;
[if (c.superClass -> size() > 0) ]
public class [c.name /] extends [c.superClass.name /] {
[else]
public class [c.name /] extends Organization {
[/if]
   //Constructor
   public [c.name /] (String name, Environment env, AgentRole agRole, Organization org) {
     super(name, env, agRole);

     [for(b : Belief | c.ownedBelief)]
       Belief [b.name.concat('B')/] = new Belief("[b.name.concat('B')/]", "[b.type/]",
       "[b.default/]");
          addBelief("[b.name.concat('B')/]", [b.name.concat('B')/]);
        [/for]

     [for(g : Goal | c.ownedGoal)]
       Goal [g.name.concat('G')/] = new LeafGoal("[g.name.concat('G')/]", "[g.type/]",
       "[g.default/]");
       addGoal("[g.name.concat('G')/]", [g.name.concat('G')/]);
     [/for]

    [for(a : ActionClass | c.ownedAction)]
      [if (a.actionSemantics.toString().trim().size() <= 0)]
       Action [a.name/]Ac = new Action("[a.name/]", null, null);
       addAction("[a.name/]", [a.name/]Ac);
      [/if]
    [/for]

    [for(p : PlanClass | c.ownedPlan)]
      Plan [p.name.concat('Plan')/] = new [p.name/]("[p.name.concat('Plan')/]",
        [p.owendGoal.name/]G);
      addPlan("[p.name.concat('Plan')/]", [p.name.concat('Plan')/]);
      [for(ac : ActionClass | p.ownedAction)]
        [p.name.concat('Plan')/].addAction("[ac.name.concat('Ac')/]",
       [ac.name.concat('Ac')/]);
      [/for]
    [/for]
   }
}
[/file]
[/template]
```

Figura 23 Template para Organization.

Cada arquivo .mtl criado contém o *template* da geração de cada entidade em um projeto do tipo Acceleo, os quais possibilitam a geração de código a partir dos modelos desenhados em MAS-ML *Tool*. Estes *templates* foram desenvolvidos para o *framework* JAMDER, porém, caso o usuário queira gerar para outros *frameworks* ou outra linguagem de programação, é necessário apenas adaptar os *templates* específicos para este *framework* ou linguagem escolhida.

O código em JAMDER, que os templates geram, compilam corretamente desde que as entidades e propriedades das entidades tenham nomenclatura distintas, a fim de evitar a criação de duas entidades ou propiedades com a mesma nomenclatura.

6 ESTUDO DE CASO – SISTEMA MOODLE

Neste capítulo é ilustrada a geração de código para desenvolvimento de um SMA para o ambiente de aprendizagem colaborativa *Moodle* [MOODLE, 2011]. O Moodle é um Sistema *Open Source* de Gerenciamento de Cursos - *Course Management System* (CMS), também conhecido como *Learning Management System* (LMS) ou um Ambiente Virtual de Aprendizagem (AVA), que representa um meio de interação entre alunos e professores em um ambiente virtual. Tornou-se muito popular entre os educadores de todo o mundo como uma ferramenta para criar *websites* dinâmicos para seus alunos. O foco do projeto *Moodle* é disponibilizar aos educadores as melhores ferramentas para gerenciar e promover a aprendizagem [MOODLE, 2011]. Em resumo, ele permite que alunos, professores e colaboradores possam integrar-se em um ambiente virtual através de cursos on-line.

O intuito do estudo de caso é ilustrar a aplicação da abordagem proposta através da criação de um sistema multi-agentes para o *Moodle*, identificando os agentes, seus papéis, a organização e o ambiente que habitam através da modelagem na ferramenta MAS-ML *Tool*. Após a modelagem ser estabelecida, o próximo passo é a geração de código a partir do modelo proposto. Inicialmente, é necessário criar um projeto do tipo Acceleo na ferramenta Eclipse contendo as classes e os *templates* (arquivo .mtl) d Acceleo designados para gerar código em JAMDER.

Para executar a geração de código do protótipo deste projeto SMA, foi utilizado o diagrama de organização de MAS-ML *Tool*. Ele contém as entidades: agentes, organização, papel de agente e ambiente e os relacionamentos *play*, *ownership* e *inhabit*, entre estas entidades. Cada *template* será executado individualmente utilizando o diagrama que contenha a entidade necessária ao *template* a fim de gerá-las.

A representação do ambiente para este projeto possui apenas uma instância de *Environment* de JAMDER, onde esta instância contém as outras entidades, aqui representado como *MoodleEnv*. Tendo o conhecimento de que o objetivo do sistema *Moodle* é aproximar discentes e docentes virtualmente, a representação da organização é de apenas uma instância de *Organization* de JAMDER, no caso,

representada por *MoodleOrg*. Devido a estas características, todos os agentes e papéis deste SMA estão em uma mesma organização e consequentemente em um mesmo ambiente.

A Figura 24 contém as entidades dos agentes, organização, papéis de agente e ambiente. Porém, todas as propriedades das entidades como crenças, objetivos, entre outros, foram omitidas neste diagrama, devido ao limite de espaço para exibição no texto do trabalho, contudo, a estrutura das entidades possuem as propriedades que serão detalhadas nas próximas seções.

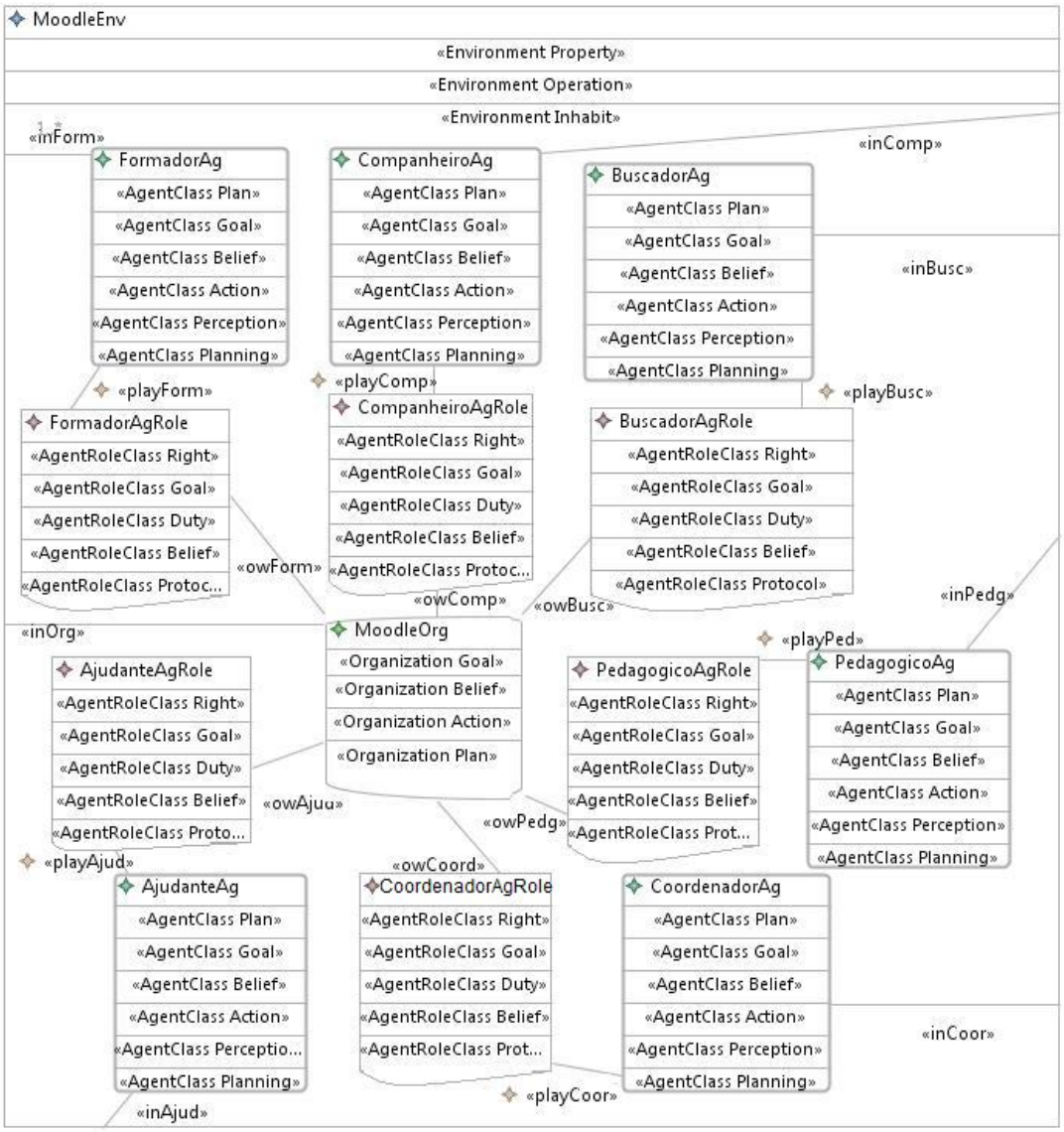


Figura 24 Modelagem do protótipo do sistema Moodle modelado em MAS-ML *Tool*.

Um ponto importante a observar neste protótipo de SMA é que as crenças dos agentes e papéis de agente foram representadas na modelagem como arquivos .pl, onde estes arquivos contêm as informações das crenças para serem lidas pela classe gerada. A função deste arquivo é apenas conter as crenças e serão obtidas as suas informações a partir de sua leitura, adaptando o código após ser gerado pela ferramenta.

A seguir, os detalhes das entidades de agente e papel de agente envolvidas neste protótipo de SMA são apresentados.

6.1 Agentes

Tendo como base a ideia do sistema *Moodle*, é possível presumir alguns agentes necessários onde cada um deles é representado por um tipo de agente definido em MAS-ML 2.0. Partindo desta premissa, foram identificados os seguintes agentes para este ambiente:

6.1.1 Agente companheiro aprendizagem

Um agente deste tipo deve ser capaz de escolher autonomamente dentre uma gama pré-estabelecida de estratégias de interação afetiva, tais como mensagens de apoio. Apresenta mensagens de incentivo (reforço positivo) quando o usuário, através das interações manifestadas, der indícios de que não apresenta dificuldades para acompanhar as discussões e/ou as tarefas propostas e/ou o conteúdo, e até mesmo quando o aluno impõe um ritmo muito superior à média de sua turma ou grupo de trabalho. Devido à necessidade de guardar notas da turma para comparação e enviar mensagens rapidamente, este agente é caracterizado como um agente reativo baseado em conhecimento. A sua estrutura é dada a seguir e ilustrada pela Figura 25.

Este agente deve conhecer as estratégias de interação afetiva e as mensagens de apoio (mensagens de cunho afetivo) que serão dadas ao usuário mediante suas crenças. As crenças estão contidas em um arquivo .pl.

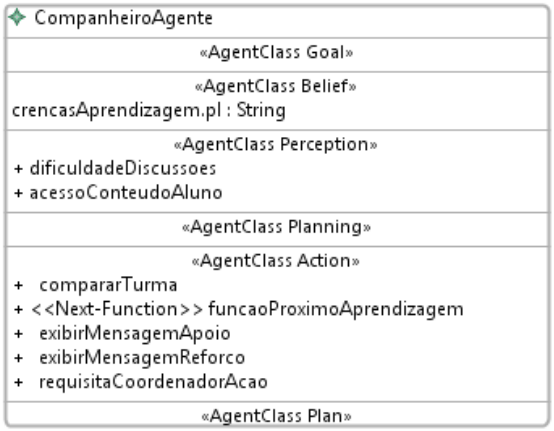


Figura 25 Modelagem de *CompanheiroAgente*

Quando o usuário, através das percepções das interações manifestadas, apresentar indícios de dificuldade para acompanhar as discussões e/ou as tarefas propostas e/ou o conteúdo, o atributo de percepção é identificado por *dificuldadeDiscussoes* e pode ser obtido através de carinhas do *chat*. A percepção de *acessoConteudoAluno* é designada para saber se o aluno está acessando os conteúdos.

A ação *compararTurma* compara com a turma e verifica se o aluno está muito superior ou muito inferior, no entanto, antes desta comparação, é necessário que a precondição *mediaAluno* seja diferente de nulo. A ação *exibirMensagemApoio* exibe a mensagem de apoio, mas a pré-condição *alunoSemDificuldade* precisa ser satisfeita, ou seja, o aluno precisa estar entendendo o conteúdo. A ação *exibirMensagemReforco* surge como efeito oposto, onde a pré-condição *alunoComDificuldade* deve ser verdadeira. A ação *requisitaCoordenadorAcao* requisita uma ação de outro agente que entre em contato com o coordenador para satisfazer as precondições de *mensagensDicasCurso* e *dificuldadeFuncionalidades*, entre outros. As pré-condições e poscondições em MAS-ML *Tool* não puderam ser representadas pois a ferramenta ainda não disponibiliza esta funcionalidade.

De acordo com a Figura 26, o código gerado para este agente possui em seu construtor a chamada à sua super classe, neste caso, ModelAgent. Logo após, há a criação das instâncias das ações referentes a este agente, a chamada ao método *addAction (String, Action)*, onde informa que estas instâncias estão sendo inclusas na lista de ações do agente e por fim, as percepções definidas. Os métodos criados

para este agente foram o *nextFunction (Belief, Perception)* que chama seu método correspondente *funcaoProximoAprendizagem (Belief, Perception)*.

```
import jamder.behavioural.*;
import jamder.Environment;
import jamder.roles.AgentRole;
import jamder.structural.*;
import java.util.List;
import jamder.agents.*;

public class CompanheiroAgente extends ModelAgent {
   //Constructor
   public CompanheiroAgente (String name, Environment env, AgentRole agRole) {
     super(name, env, agRole);
     addBelief("crencasAprendizagem.pl", new Belief("crencasAprendizagem.pl", "String", ""));

     Action compararTurmaAc = new Action("compararTurmaAc", null, null);
     addAction("compararTurmaAc", compararTurmaAc);
     Action exibirMensagemApoioAc = new Action("exibirMensagemApoioAc", null, null);
     addAction("exibirMensagemApoioAc", exibirMensagemApoioAc);
     Action exibirMensagemReforcoAc = new Action("exibirMensagemReforcoAc", null, null);
     addAction("exibirMensagemReforcoAc", exibirMensagemReforcoAc);
     Action requisitaCoordenadorAcaoAc = new Action("requisitaCoordenadorAcaoAc", null,
       null);
     addAction("requisitaCoordenadorAcaoAc", requisitaCoordenadorAcaoAc);

     addPerceive("dificuldadeDiscussoes", null);
     addPerceive("acessoConteudoAluno", null);
   }

   protected Belief nextFunction(Belief belief, String perception) {
       return funcaoProximoAprendizagem(belief, perception);
   }
   private Belief funcaoProximoAprendizagem(Belief belief, String perception) {
       return null;
   }

}
```

Figura 26 Classe JAMDER CompanheiroAgente gerada.

6.1.2 Assistente de aprendizagem

Este agente deve ser capaz de acompanhar o usuário nos diferentes cursos, disciplinas e projetos em que este participar no intuito de contribuir com o usuário através de dicas, sugestões e mensagens relacionadas à temática em curso e não apenas mensagens de cunho afetivo (apoio). Ele é um agente baseado em objetivo com planejamento, pois necessita montar um plano de estudos, sugerindo cursos e disciplinas para o aluno tendo como base os cursos e displinas que o aluno está fazendo. A Figura 27 mostra o modelo deste agente.

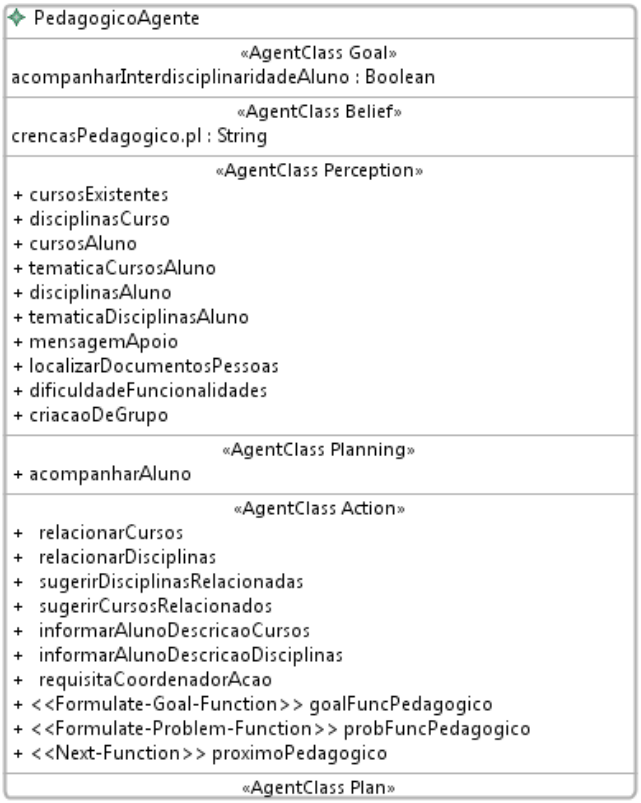


Figura 27 Modelagem de *PedagogicoAgente*

O agente pedagógico percebe os cursos existentes na Instituição de Ensino, percebe as disciplinas do(s) curso(s) que o aluno está matriculado e percebe em quais cursos o aluno está matriculado. A partir das percepções, as ações pertencentes ao agente incluem relacionar ou identificar os cursos, identificar as disciplinas e sugerir outras disciplinas ou cursos relacionados ao curso que o aluno frequenta. Este agente também fornece informações sobre as disciplinas e cursos em que o aluno está matriculado. O agente pode enviar mensagens ao agente coordenador solicitando que outros agentes realizem outras tarefas, dependendo de qual precondição seja satisfeita, como enviar mensagens de apoio, entre outras.

Assim como um agente baseado em objetivo com planejamento, os métodos pertencentes a este tipo de agente necessitam ser implementados pelo desenvolvedor, por exemplo, para montar seu plano em tempo de execução, a fim de alcançar seu objetivo, onde o plano consiste em *acompanharAluno*.

A Figura 28 exibe o código gerado para este agente.

```
import jamder.behavioural.*;
import jamder.Environment;
import jamder.roles.AgentRole;
import jamder.structural.*;
import java.util.List;
import jamder.agents.*;

public class PedagogicoAgente extends GoalAgent {

   //Constructor
   public PedagogicoAgente (String name, Environment env, AgentRole agRole) {
   super(name, env, agRole);

     addBelief("crencasPedagogico.pl", new Belief("crencasPedagogico.pl", "String", ""));

     Goal acompanharInterdisciplinaridadeAlunoG = new
       LeafGoal("acompanharInterdisciplinaridadeAlunoG", "Boolean", "");
     addGoal("acompanharInterdisciplinaridadeAlunoG", acompanharInterdisciplinaridadeAlunoG);

     Action relacionarCursosAc = new Action("relacionarCursosAc", null, null);
     addAction("relacionarCursosAc", relacionarCursosAc);
     Action relacionarDisciplinasAc = new Action("relacionarDisciplinasAc", null, null);
     addAction("relacionarDisciplinasAc", relacionarDisciplinasAc);
     Action sugerirDisciplinasRelacionadasAc = new Action("sugerirDisciplinasRelacionadasAc",
       null, null);
     addAction("sugerirDisciplinasRelacionadasAc", sugerirDisciplinasRelacionadasAc);
     Action sugerirCursosRelacionadosAc = new Action("sugerirCursosRelacionadosAc", null,
       null);
     addAction("sugerirCursosRelacionadosAc", sugerirCursosRelacionadosAc);
     Action informarAlunoDescricaoCursos Ac = new Action("informarAlunoDescricaoCursos Ac",
       null, null);
     addAction("informarAlunoDescricaoCursos Ac", informarAlunoDescricaoCursos Ac);
     Action informarAlunoDescricaoDisciplinas Ac = new
       Action("informarAlunoDescricaoDisciplinas Ac", null, null);
     addAction("informarAlunoDescricaoDisciplinas Ac", informarAlunoDescricaoDisciplinas Ac);
     Action requisitaCoordenadorAcaoAc = new Action("requisitaCoordenadorAcaoAc", null,
       null);
     addAction("requisitaCoordenadorAcaoAc", requisitaCoordenadorAcaoAc);

     addPerceive("cursosExistentes", null);
     addPerceive("disciplinasCurso", null);
     addPerceive("cursosAluno", null);
     addPerceive("tematicaCursosAluno", null);
     addPerceive("disciplinasAluno", null);
     addPerceive("tematicaDisciplinasAluno", null);
     addPerceive("mensagemApoio", null);
     addPerceive("localizarDocumentosPessoas", null);
     addPerceive("dificuldadeFuncionalidades", null);
     addPerceive("criacaoDeGrupo ", null);

     Plan acompanharAlunoPlan = new Plan("acompanharAluno Plan", null);
     addPlan("acompanharAluno Plan", acompanharAluno Plan);
   }

   protected Plan planning(List<Action> actions){
      return null;
   }

   protected Goal formulateGoalFunction(Belief belief) {
      return goalFuncPedagogico(belief);
   }
   private Goal goalFuncPedagogico(Belief belief) {
      return null;
   }
   protected List<Action> formulateProblemFunction(Belief belief, Goal goal) {
      return probFuncPedagogico(belief, goal);
   }
   private List<Action> probFuncPedagogico(Belief belief, Goal goal) {
      return null;
   }
   protected Belief nextFunction(Belief belief, String perception) {
      return proximoPedagogico(belief, perception);
   }
   private Belief proximoPedagogico(Belief belief, String perception) {
      return null;
```

```
    }

    public void percept(String perception) { }
}
```

Figura 28 Classe JAMDER PedagogicoAgente gerada.

6.1.3 Agente buscador de informações

Este tipo de agente localiza pessoas dentro do ambiente *Moodle* que estejam envolvidas com disciplinas relacionadas a um determinado tema de interesse. Uma variação deste agente buscador poderia se envolver na localização de documentos (páginas, projetos e outros objetos digitais) que envolvam um determinado tema de interesse. O intuito deste agente é fazer pesquisa contínua e, autonomamente, mostrar documentos e contatos de pessoas sempre que localizar interesses comuns em potencial. A Figura 29 mostra a estrutura deste agente.

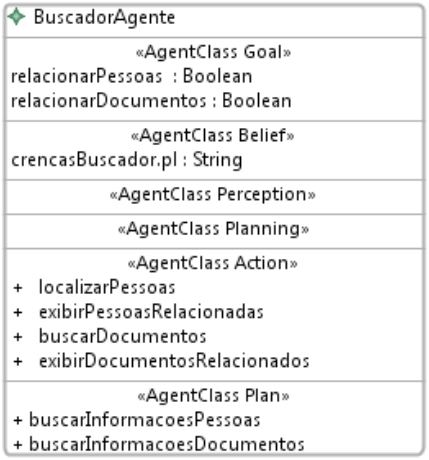


Figura 29 Modelagem de *BuscadorAgente*

As crenças deste agente armazenam quais são as pessoas, grupos e disciplinas através do arquivo *crencasBuscador.pl*. O objetivo *relacionarPessoas* verifica as pessoas no ambiente *Moodle* que estejam envolvidas com projetos ou disciplinas em comum. O objetivo *relacionarDocumentos* busca documentos como páginas, projetos ou outro arquivo digital que tenham em comum, por exemplo, sua palavra chave.

As ações pertencentes a este agente podem ser de quatro formas. A ação *localizarPessoa*, como o próprio nome diz, localiza as pessoas que estejam relacionadas ao mesmo tema do usuário. A ação *exibirPessoasRelacionadas* é

utilizada para exibir o resultado, onde a pré-condição desta ação é que o resultado da localização de outras pessoas seja diferente de nulo. Na mesma linha de raciocínio, as ações *buscarDocumentos* e *exibirDocumentosRelacionados* são utilizadas para os documentos.

Devido a este agente já saber o que fazer e como deve fazer, ele é classificado como um agente baseado em objetivo com plano. Neste caso, ele possui dois planos pré-definidos. O plano *buscarInformacoesPessoas* tem duas ações, *localizarPessoa* e *exibirPessoasRelacionadas*, nesta ordem, para atingir o objetivo *relacionarPessoas*. Também na mesma linha de raciocínio, o plano *buscarInformacoesDocumentos* utiliza as ações *localizarDocumentos* e *exibirDocumentosRelacionados*, nesta ordem, para atingir o objetivo *relacionarDocumentos*.

A Figura 30 exibe o código gerado para este agente.

```
import jamder.behavioural.*;
import jamder.Environment;
import jamder.roles.AgentRole;
import jamder.structural.*;
import java.util.List;
import jamder.agents.*;

public class BuscadorAgente extends MASMLAgent {
   //Constructor
   public BuscadorAgente (String name, Environment env, AgentRole agRole) {
   super(name, env, agRole);
     addBelief("crencasBuscador.pl", new Belief("crencasBuscador.pl", "String", ""));

     Goal relacionarPessoasG = new LeafGoal("relacionarPessoasG", "Boolean", "false");
     addGoal("relacionarPessoasG", relacionarPessoasG);
     Goal relacionarDocumentosG = new LeafGoal("relacionarDocumentosG", "Boolean", "false");
     addGoal("relacionarDocumentosG", relacionarDocumentosG);

     Action localizarPessoasAc = new Action("localizarPessoasAc", null, null);
     addAction("localizarPessoasAc", localizarPessoasAc);
     Action exibirPessoasRelacionadasAc = new Action("exibirPessoasRelacionadasAc", null,
       null);
     addAction("exibirPessoasRelacionadasAc", exibirPessoasRelacionadasAc);
     Action buscarDocumentosAc = new Action("buscarDocumentosAc", null, null);
     addAction("buscarDocumentosAc", buscarDocumentosAc);
     Action exibirDocumentosRelacionadosAc = new Action("exibirDocumentosRelacionadosAc",
       null, null);
     addAction("exibirDocumentosRelacionadosAc", exibirDocumentosRelacionadosAc);

     Plan buscarInformacoesPessoasPlan = new Plan("buscarInformacoesPessoasPlan",
       relacionarDocumentosG);
     addPlan("buscarInformacoesPessoasPlan", buscarInformacoesPessoasPlan);
     Plan buscarInformacoesDocumentosPlan = new Plan("buscarInformacoesDocumentosPlan",
       relacionarPessoasG);
     addPlan("buscarInformacoesDocumentosPlan", buscarInformacoesDocumentosPlan);
   }
   public void percept(String perception) { }
}
```

Figura 30 Classe JAMDER BuscadorAgente gerada.

As ações *localizarPessoa* e *exibirPessoasRelacionadas* do plano *buscarInformacoesPessoas* e as ações *localizarDocumentos* e *exibirDocumentosRelacionados* do plano *buscarInformacoesDocumentos* foram inclusas manualmente em seus respectivos planos após a geração desta classe pois, a ferramenta MAS-ML *Tool*, na versão atual, ainda não possui a associação entre plano e ações semanticamente.

6.1.4 Agente que fornece ajuda sobre o Moodle

Este agente é do tipo reativo simples e possui uma lista de várias percepções sobre as dificuldades que o usuário está tendo, e diante disto, escolhe a ação correspondente. Este agente percebe em que momento o usuário está e ao mesmo tempo oferece autonomamente dicas sobre como fazer o melhor uso de uma determinada funcionalidade, mais especificamente, a ação para uma determinada tarefa. A Figura 31 mostra que apenas as percepções e ações deste agente é que o torna um agente reativo simples.

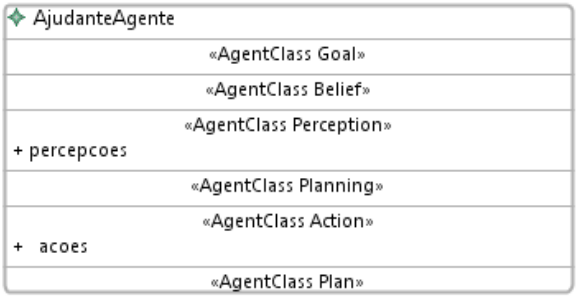


Figura 31 Modelagem de *AjudanteAgente*

Devido ao limite de tamanho da figura, algumas das percepções e ações propostas para este agente serão definidas a seguir:

- *percepcaoForuns;*
- *percepcaoParticipantes;*
- *percepcaoCalendario;*
- *percepcaoCalendarioNovoEvento;*
- *percepcaoAtividadeRecenteTodosParticipantes*
- *percepcaoAtivarEdicao;*

- *percepcaoConfiguracoes;*
- *percepcaoDesignarFuncoes*
- *percepcaoRelatorioDeNotas.*

Para corresponder à percepção, o agente executa uma das ações para atender à percepção. Ressaltando que estas ações não possuem pré ou poscondições. As ações são exibidas a seguir:

- *exibirDicasForuns;*
- *exibirDicasParticipantes;*
- *exibirDicasCalendario;*
- *exibirDicasCalendarioNovoEvento;*
- *exibirDicasAtividadeRecenteTodosParticipantes;*
- *exibirDicasAtivarEdicao;*
- *exibirDicasConfiguracoes;*
- *exibirDicasDesignarFuncoes;*
- *exibirDicasRelatorioDeNotas;*

Diferentemente das outras ações, a ação *requisitaCoordenadorAcao* tem a finalidade de escolher outro agente para fazer alguma tarefa e possui as seguintes pré-condições: *mensagemApoio*, *mensagensDicasCurso*, *criacaoDeGrupo, localizarDocumentosPessoas* e *dificuldadeFuncionalidades*. Estas pré-condições devem ser satisfeitas para que esta ação seja executada.

A Figura 32 exibe o código gerado para o agente *AjudanteAgente*.

```
import jamder.behavioural.*;
import jamder.Environment;
import jamder.roles.AgentRole;
import jamder.structural.*;
import java.util.List;
import jamder.agents.*;

public class AjudanteAgente extends ReflexAgent {

   //Constructor
   public AjudanteAgente (String name, Environment env, AgentRole agRole) {
   super(name, env, agRole);

   Action exibirDicasForuns = new Action("exibirDicasForuns", null, null);
   Action exibirDicasParticipantes = new Action("exibirDicasParticipantes", null, null);
   Action exibirDicasCalendario = new Action("exibirDicasCalendario", null, null);
```

```
    Action exibirDicasCalendarioNovoEvento = new Action("exibirDicasCalendarioNovoEvento",
        null, null);
    Action exibirDicasAtividadeRecenteTodosParticipantes = new
        Action("exibirDicasAtividadeRecenteTodosParticipantes", null, null);
    Action exibirDicasAtivarEdicao = new Action("exibirDicasAtivarEdicao", null, null);
    Action exibirDicasConfiguracoes = new Action("exibirDicasConfiguracoes", null, null);
    Action exibirDicasDesignarFuncoes = new Action("exibirDicasDesignarFuncoes", null, null);
    Action exibirDicasRelatorioDeNotas = new Action("exibirDicasRelatorioDeNotas", null,
        null);

    // Adding preconditions
    requisitaCoordenadorAcao.addPreCondition("mensagemApoio", new Condition());
    requisitaCoordenadorAcao.addPreCondition("mensagensDicasCurso", new Condition());
    requisitaCoordenadorAcao.addPreCondition("localizarDocumentosPessoas", new Condition());
    requisitaCoordenadorAcao.addPreCondition("dificuldadeFuncionalidades", new Condition());
    requisitaCoordenadorAcao.addPreCondition("criacaoDeGrupo", new Condition());

    addAction("exibirDicasForuns", exibirDicasForuns);
    addAction("exibirDicasParticipantes", exibirDicasParticipantes);
    addAction("exibirDicasCalendario", exibirDicasCalendario);
    addAction("exibirDicasCalendarioNovoEvento", exibirDicasCalendarioNovoEvento);
    addAction("exibirDicasAtividadeRecenteTodosParticipantes",
        exibirDicasAtividadeRecenteTodosParticipantes);
    addAction("exibirDicasAtivarEdicao", exibirDicasAtivarEdicao);
    addAction("exibirDicasConfiguracoes", exibirDicasConfiguracoes);
    addAction("exibirDicasDesignarFuncoes", exibirDicasDesignarFuncoes);
    addAction("exibirDicasRelatorioDeNotas", exibirDicasRelatorioDeNotas);

    addPerceive("exibirDicasForuns", null);
    addPerceive("exibirDicasParticipantes", null);
    addPerceive("exibirDicasCalendario", null);
    addPerceive("exibirDicasCalendarioNovoEvento", null);
    addPerceive("exibirDicasAtividadeRecenteTodosParticipantes", null);
    addPerceive("exibirDicasAtivarEdicao", null);
    addPerceive("exibirDicasConfiguracoes", null);
    addPerceive("exibirDicasDesignarFuncoes", null);
    addPerceive("exibirDicasRelatorioDeNotas", null);

    }
}
```

Figura 32 Classe JAMDER AjudanteAgente gerada.

Após a geração do código, é necessário que o desenvolvedor forneça as informações finais para este agente tal como a ação correspondente à percepção. Neste caso foi colocado manualmente um exemplo das pré-condições de JAMDER após a geração da classe *AjudanteAgente*, pois a ferramenta MAS-ML *Tool* na sua versão atual ainda não fornece estas informações.

6.1.5 Agente formador de grupos

Este agente deve ser capaz de auxiliar, autonomamente, os usuários, alunos e formadores, na composição de grupos de trabalho levando em conta afinidade de temas ou de perfis de aprendizagem. Para isto deve considerar determinados critérios estabelecidos por um formador de uma ou mais turmas, ou pelo próprio usuário interessado em integrar-se a grupos de trabalho. A Figura 33 mostra a estrutura do agente *FormadorAgente*.

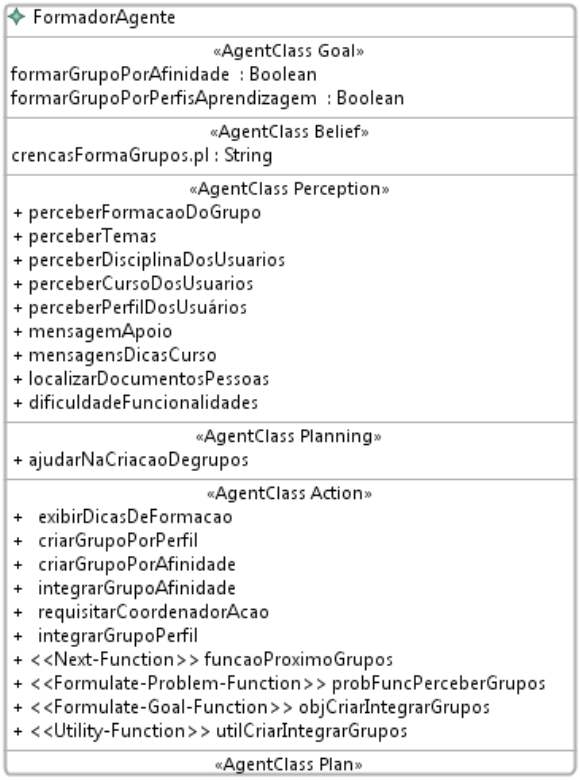


Figura 33 Modelagem de *FormadorAgente*

As crenças deste agente armazenam os cursos por tema e os perfis de aprendizagem do professor ou usuário. Suas ações propostas incluem:

- *exibirDicasDeFormacao*: exibe dicas ou sugestões para a criação de um grupo relacionado ao tema do usuário;
- *criarGrupoPorPerfil*: cria grupos baseados no perfil do usuário;
- *criarGrupoPorAfinidade*: cria grupos baseados no tema;
- *integrarGrupoAfinidade*: integra grupos baseados no tema;
- *requisitarCoordenadorAcao*: requisita ao coordenador alguma tarefa baseada na satisfação de alguma precondição, quando verdadeira;

O planejamento *ajudarNaCriacaoDegrupos* é montado em tempo de execução do agente e possui dois objetivos, *formarGrupoPorPerfisAprendizagem* e *formarGrupoPorAfinidade*. Os dois formam grupos, mas a diferença está na premissa utilizada para planejar o melhor meio e, consequentemente, sugerir formas para se criar ou integrar grupos, seguindo um tema ou perfil dos usuários que se

enquadram ao grupo a ser definido. Esta escolha mostra que em tempo de execução um dos dois objetivos é escolhido para ser atingido.

Como um agente baseado em utilidade, este possui os métodos definidos em MAS-ML 2.0 e são determinados pelos métodos:

- <<*next-function*>> *funcaoProximoGrupos*: dá sugestões de criação ou integração de grupos;
- <<*formule-problem-function*>> *probFuncPerceberGrupos*: reconhece a possível formação de grupo segundo um tema ou perfil sugerido;
- <<*formule-goal-function*>> *objCriarIntegrarGrupos*: auxilia o usuário na formação de grupos temáticos de assunto, dando assim um maior valor ao vínculo de colaboração entre os integrantes do grupo a fim de proporcionar melhor a aprendizagem;
- <<*utility-function*>> *utilCriarIntegrarGrupos*: cria um equilíbrio entre a formação de temas e perfis;

A Figura 34 exibe o código gerado para este agente.

```
import jamder.behavioural.*;
import jamder.Environment;
import jamder.roles.AgentRole;
import jamder.structural.*;
import java.util.List;
import jamder.agents.*;

public class FormadorAgente extends UtilityAgent {
   //Constructor
  public FormadorAgente (String name, Environment env, AgentRole agRole) {
    super(name, env, agRole);
    addBelief("crencasFormaGrupos.pl", new Belief("crencasFormaGrupos.pl", "String", ""));

    Goal formarGrupoPorAfinidadeG = new LeafGoal("formarGrupoPorAfinidadeG", "Boolean",
        "false");
    addGoal("formarGrupoPorAfinidadeG", formarGrupoPorAfinidadeG);
    Goal formarGrupoPorPerfisAprendizagemG = new
       LeafGoal("formarGrupoPorPerfisAprendizagemG", Boolean, "");
    addGoal("formarGrupoPorPerfisAprendizagemG", formarGrupoPorPerfisAprendizagemG);

    Action exibirDicasDeFormacaoAc = new Action("exibirDicasDeFormacaoAc", null, null);
    addAction("exibirDicasDeFormacaoAc", exibirDicasDeFormacaoAc);
    Action criarGrupoPorPerfilAc = new Action("criarGrupoPorPerfilAc", null, null);
    addAction("criarGrupoPorPerfilAc", criarGrupoPorPerfilAc);
    Action criarGrupoPorAfinidadeAc = new Action("criarGrupoPorAfinidadeAc", null, null);
    addAction("criarGrupoPorAfinidadeAc", criarGrupoPorAfinidadeAc);
    Action integrarGrupoAfinidadeAc = new Action("integrarGrupoAfinidadeAc", null, null);
    addAction("integrarGrupoAfinidadeAc", integrarGrupoAfinidadeAc);
    Action requisitarCoordenadorAcaoAc = new Action("requisitarCoordenadorAcaoAc", null,
        null);
    addAction("requisitarCoordenadorAcaoAc", requisitarCoordenadorAcaoAc);
    Action integrarGrupoPerfilAc = new Action("integrarGrupoPerfilAc", null, null);
    addAction("integrarGrupoPerfilAc", integrarGrupoPerfilAc);

    addPerceive("perceberFormacaoDoGrupo", null);
    addPerceive("perceberTemas", null);
```

```
    addPerceive("perceberDisciplinaDosUsuarios", null);
    addPerceive("perceberCursoDosUsuarios", null);
    addPerceive("perceberPerfilDosUsuarios", null);
    addPerceive("mensagemApoio", null);
    addPerceive("mensagensDicasCurso", null);
    addPerceive("localizarDocumentosPessoas", null);
    addPerceive("dificuldadeFuncionalidades ", null);

    Plan ajudarNaCriacaoDegruposPlan = new Plan("ajudarNaCriacaoDegruposPlan", null);
    addPlan("ajudarNaCriacaoDegruposPlan", ajudarNaCriacaoDegruposPlan);
  }

  protected Plan planning(List<Action> actions){
        return null;
  }

  protected Belief nextFunction(Belief belief, String perception) {
        return funcaoProximoGrupos(belief, perception);
  }
  private Belief funcaoProximoGrupos(Belief belief, String perception) {
        return null;
  }
  protected List<Action> formulateProblemFunction(Belief belief, Goal goal) {
        return probFuncPerceberGrupos(belief, goal);
  }
  private List<Action> probFuncPerceberGrupos(Belief belief, Goal goal) {
        return null;
  }
  protected Goal formulateGoalFunction(Belief belief) {
        return objCriarIntegrarGrupos(belief);
  }
  private Goal objCriarIntegrarGrupos(Belief belief) {
        return null;
  }
  protected Integer utilityFunction(Action action) {
        return utilCriarIntegrarGrupos(action);
  }
  private Integer utilCriarIntegrarGrupos(Action action) {
        return 0;
  }
  public void percept(String perception) { }
}
```

Figura 34 Classe JAMDER FormadorAgente gerada.

6.1.6 Agente Coordenador

Este tipo de agente deve ser capaz de centralizar as requisições dos agentes e as informações que eles enviam uns aos outros, fazendo deste agente um intermediador entre os outros agentes. O objetivo do agente coordenador é requisitar as ações dos agentes (*requisitarAcoesDoAgente*). Para atingir este objetivo, o plano *requisitarAcaoPlano* usa duas ações: *verificaAcaoAgente* e *requisitaAcao*, onde a primeira checa se o agente é capaz de realizar a tarefa e a segunda solicita ao agente que execute a ação depois de verificar se o agente pode executar (*verificacaAcaoAgente*). A Figura 35 mostra a estrutura do agente *CoordenadorAgente*.

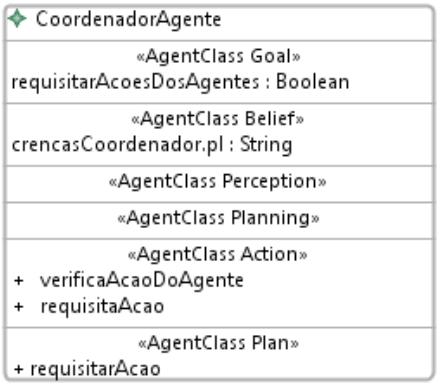


Figura 35 Modelagem de CoordenadorAgente

A Figura 36 exibe o código gerado para o agente CoordenadorAgente.

```
import jamder.behavioural.*;
import jamder.Environment;
import jamder.roles.AgentRole;
import jamder.structural.*;
import java.util.List;
import jamder.agents.*;

public class CoordenadorAgente extends MASMLAgent {
   //Constructor
   public CoordenadorAgente (String name, Environment env, AgentRole agRole) {
     super(name, env, agRole);
     addBelief("crencasCoordenador.pl", new Belief("crencasCoordenador.pl", "String", ""));
     Goal requisitarAcoesDosAgentesG = new LeafGoal("requisitarAcoesDosAgentesG", "Boolean",
       "false");
     addGoal("requisitarAcoesDosAgentesG", requisitarAcoesDosAgentesG);

     Action verificaAcaoDoAgenteAc = new Action("verificaAcaoDoAgenteAc", null, null);
     addAction("verificaAcaoDoAgenteAc", verificaAcaoDoAgenteAc);
     Action requisitaAcaoAc = new Action("requisitaAcaoAc", null, null);
     addAction("requisitaAcaoAc", requisitaAcaoAc);

     Plan requisitarAcaoPlan = new Plan("requisitarAcaoPlan", requisitarAcoesDosAgentesG);
     addPlan("requisitarAcaoPlan", requisitarAcaoPlan);

     requisitarAcaoPlan.addAction("verificaAcaoDoAgenteAc", verificaAcaoDoAgenteAc);
     requisitarAcaoPlan.addAction("requisitaAcaoAc", requisitaAcaoAc);
   }

public void percept(String perception) { }
}
```

Figura 36 Classe JAMDER CoordenadorAgente gerada.

Neste exemplo de código gerado, as ações foram inseridas manualmente, pois a atual versão da ferramenta MAS-ML ainda não fornece a inclusão da informação de ações em sua estrutura. A saber: *verificaAcaoAgente* e *requisitaAcao*.

6.2 Papéis de Agente

Em um sistema SMA, os agentes podem exercer mais de um papel de agente na mesma organização, porém, neste protótipo de SMA, cada agente possui um

papel específico. Devido a esta granularidade, as definições dos papéis de agente são as mesmas dos agentes em relação às crenças, objetivos e ações, todavia os papéis possuem outros componentes: direitos e deveres. A Figura 37 mostra mais detalhes dos papéis de agente propostos para este SMA tendo como base o diagrama de organização, no qual foram modelados.

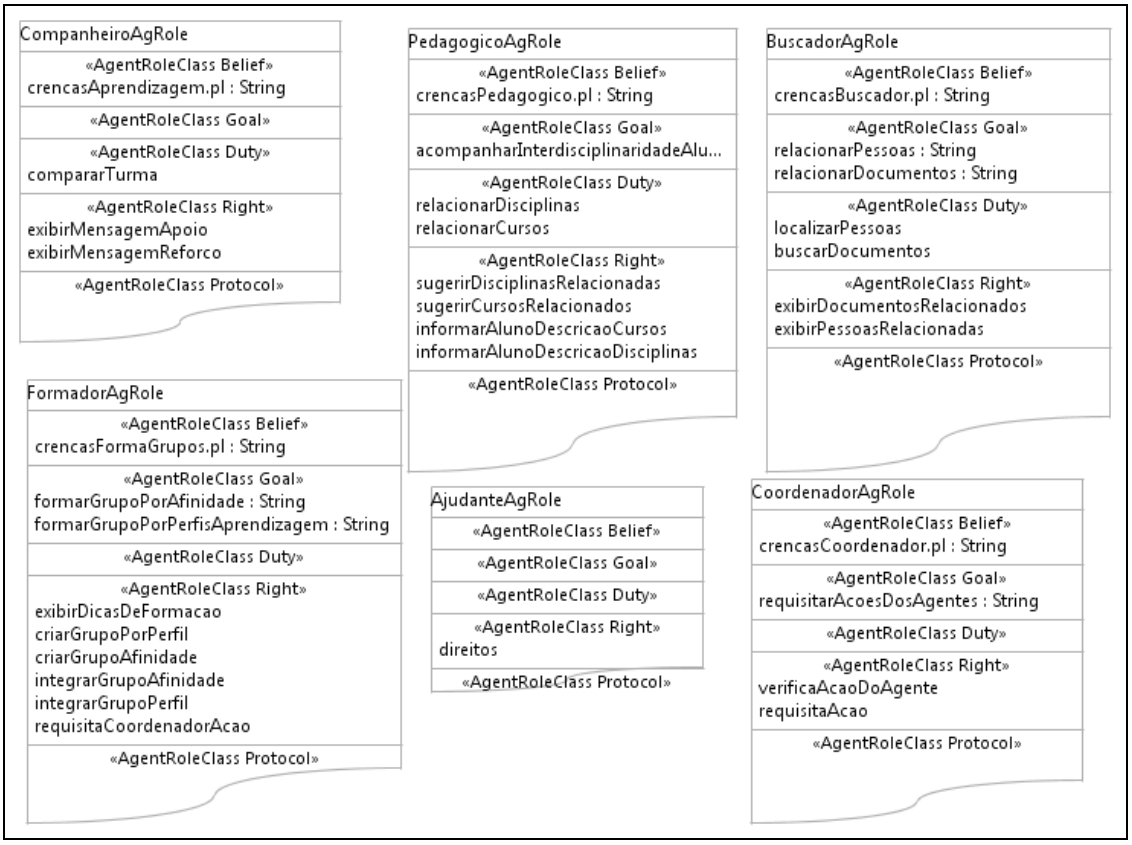


Figura 37 Parte do diagrama de organização contendo apenas os papéis de agente propostos.

A estrutura de cada papel de agente proposto para cada agente em relação aos direitos e deveres é definida a seguir:

Papel de agente companheiro de aprendizagem (*CompanheiroAgRole*) é do tipo de papel baseado em conhecimento pois necessita armazenar as notas dos alunos através de crenças. Ele contém os direitos de *exibirMensagemApoio* e *exibirMensagemReforco* pois pode ou não exibi-los e o dever de *compararTurma* pois para executar alguma ação necessita comparar a turma primeiro.

O papel de agente pedagógico (*PedagogicoAgRole*) é do tipo papel de agente proativo, pois dependendo de como está a evolução do aluno, as ações que o agente irá utilizar deste papel irão compor o plano do agente. O papel possui os deveres de *relacionarCursos* e *relacionarDisciplinas* para auxiliar a evolução do aluno. O papel também conterá os seguintes direitos:

a. *sugerirDisciplinasRelacionadas;*
b. *sugerirCursosRelacionados;*
c. *informarAlunoDescricaoCursos;*
d. *informarAlunoDescricaoDisciplinas;*

Papel de agente buscador de informações (*BuscadorAgRole*) é do tipo papel de agente proativo, pois suas ações serão utilizadas em um plano pré-definido pelo agente. Seus direitos constam em *exibirPessoasRelacionadas* e *exibirDocumentosRelacionados* para exibir as pessoas ou documentos relacionados ao interesse do usuário corrente. Seus deveres *localizarPessoas* e *buscarDocumentos* identificam a obrigação de localizar as pessoas ou documentos relacionados ao usuário corrente.

Papel de agente ajudante do Moodle (*AjudanteAgRole*) é do tipo papel de agente reativo simples, portanto, não possui crenças nem objetivos. Ele fornece ajuda sobre o funcionamento do Moodle. Este papel não possui nenhum dever, mas possui os seguintes direitos:

a. *exibirDicasForuns*
b. *exibirDicasParticipantes*
c. *exibirDicasCalendario*
d. *exibirDicasCalendarioNovoEvento*
e. *exibirDicasAtividadeRecenteTodosParticipantes*
f. *exibirDicasAtivarEdicao*
g. *exibirDicasConfiguracoes*
h. *exibirDicasDesignarFuncoes*
i. *exibirDicasRelatorioDeNotas*

Papel de agente formador de grupos (*FormadorAgRole*) é do tipo papel de agente proativo, devido a possuir crenças e objetivos necessários a auxiliar os usuários na composição de grupos de trabalho. Este papel não possui nenhum dever, porém possui os seguintes direitos:

a. *exibirDicasDeFormacao*
b. *criarGrupoPorPerfil*
c. *criarGrupoAfinidade*
d. *integrarGrupoAfinidade*
e. *integrarGrupoPerfil*
f. *requisitaCoordenadorAcao*

Papel de agente coordenador (*CoordenadorAgRole*) é do tipo papel de agente proativo. Ele centraliza as requisições dos agentes e as informações que eles enviam uns aos outros, isto é, seu comportamento é um intermediador entre os outros agentes. Este papel não possui deveres, entretanto, contém os direitos *verificaAcaoDoAgente*, que pode verificar qual ação o agente está tomando, e *requisitaAcao*, que pode solicitar a outro usuário que execute uma ação.

A Figura 38 exibe o código gerado em JAMDER para os papéis de agente propostos neste protótipo de SMA.

```
import jamder.structural.*;
import jade.core.behaviours.Behaviour;
import jamder.Organization;
import jamder.agents.GenericAgent;
import jamder.roles.*;
import jamder.behavioural.*;

public class CompanheiroAgRole extends ModelAgentRole {
   //Constructor
   public CompanheiroAgRole (String name, Organization owner, GenericAgent player) {
   super(name, owner, player);
   addBelief("crencasAprendizagem.pl", new Belief("crencasAprendizagem.pl", "String", ""));
   addRight("exibirMensagemApoio", new Right());
   addRight("exibirMensagemReforco", new Right());
   addDuty("compararTurma", new Duty());
   initialize();
   }
}


import jamder.structural.*;
import jade.core.behaviours.Behaviour;
import jamder.Organization;
import jamder.agents.GenericAgent;
import jamder.roles.*;
import jamder.behavioural.*;

public class PedagogicoAgRole extends ProactiveAgentRole {
```

```
   //Constructor
   public PedagogicoAgRole (String name, Organization owner, GenericAgent player) {
   super(name, owner, player);

   addBelief("crencasPedagogico.pl", new Belief("crencasPedagogico.pl", "String", ""));

   addGoal("acompanharInterdisciplinaridadeAluno", new
       LeafGoal("acompanharInterdisciplinaridadeAluno", "String", ""));

   addRight("sugerirDisciplinasRelacionadas", new Right());
   addRight("sugerirCursosRelacionados", new Right());
   addRight("informarAlunoDescricaoCursos", new Right());
   addRight("informarAlunoDescricaoDisciplinas", new Right());

   addDuty("relacionarDisciplinas", new Duty());
   addDuty("relacionarCursos", new Duty());

   initialize();
   }
}

import jamder.structural.*;
import jade.core.behaviours.Behaviour;
import jamder.Organization;
import jamder.agents.GenericAgent;
import jamder.roles.*;
import jamder.behavioural.*;

public class BuscadorAgRole extends ProactiveAgentRole {
   //Constructor
   public BuscadorAgRole (String name, Organization owner, GenericAgent player) {
   super(name, owner, player);
   addBelief("crencasBuscador.pl", new Belief("crencasBuscador.pl", "String", ""));

   addGoal("relacionarPessoas", new LeafGoal("relacionarPessoas", "String", ""));
   addGoal("relacionarDocumentos", new LeafGoal("relacionarDocumentos", "String", ""));

   addRight("exibirDocumentosRelacionados", new Right());
   addRight("exibirPessoasRelacionadas", new Right());

   addDuty("localizarPessoas", new Duty());
   addDuty("buscarDocumentos", new Duty());

   initialize();
   }
}

import jamder.structural.*;
import jade.core.behaviours.Behaviour;
import jamder.Organization;
import jamder.agents.GenericAgent;
import jamder.roles.*;
import jamder.behavioural.*;

public class AjudanteAgRole extends AgentRole {
   //Constructor
   public AjudanteAgRole (String name, Organization owner, GenericAgent player) {
   super(name, owner, player);

   addRight("exibirDicasForuns", new Right());
   addRight("exibirDicasParticipantes", new Right());
   addRight("exibirDicasCalendario", new Right());
   addRight("exibirDicasCalendarioNovoEvento", new Right());
   addRight("exibirDicasAtividadeRecenteTodosParticipantes", new Right());
   addRight("exibirDicasAtivarEdicao", new Right());
   addRight("exibirDicasConfiguracoes", new Right());
   addRight("exibirDicasDesignarFuncoes", new Right());
   addRight("exibirDicasRelatorioDeNotas", new Right());
   addRight("exibirDicasRelatorioGeral", new Right());
   addRight("exibirDicasRelatorioDoUsuario", new Right());
   addRight("exibirDicasCategoriasItensVisaoSimples", new Right());
   addRight("exibirDicasCategoriasItensVisaoCompleta", new Right());
   addRight("exibirDicasEscalasDoCurso", new Right());
   addRight("exibirDicasLetrasVer", new Right());
   addRight("exibirDicasLetrasEditar", new Right());
```

```
   addRight("exibirDicasImportarArquivoCSV", new Right());
   addRight("exibirDicasImportarArquivoXML", new Right());
   addRight("exibirDicasExportarPlanilhaODS", new Right());
   addRight("exibirDicasExportarArquivoDeTexto", new Right());
   addRight("exibirDicasExportarPlanilhaExcel", new Right());
   addRight("exibirDicasExportarArquivoXML", new Right());
   addRight("exibirDicasConfiguracoesCurso", new Right());
   addRight("exibirDicasPreferenciasRelatorioDeNotas", new Right());
   addRight("exibirDicasGrupos", new Right());
   addRight("exibirDicasBackupDoCurso", new Right());
   addRight("exibirDicasRestaurar", new Right());
   addRight("exibirDicasImportar", new Right());
   addRight("exibirDicasReconfigurarCurso", new Right());
   addRight("exibirDicasRelatorios", new Right());
   addRight("exibirDicasPerguntas", new Right());
   addRight("exibirDicasArquivos", new Right());
   addRight("exibirDicasPerfil", new Right());

   initialize();
   }
}

import jamder.structural.*;
import jade.core.behaviours.Behaviour;
import jamder.Organization;
import jamder.agents.GenericAgent;
import jamder.roles.*;
import jamder.behavioural.*;

public class FormadorAgRole extends ProactiveAgentRole {
   //Constructor
   public FormadorAgRole (String name, Organization owner, GenericAgent player) {
   super(name, owner, player);
   addBelief("crencasFormaGrupos.pl", new Belief("crencasFormaGrupos.pl", "String", ""));

   addGoal("formarGrupoPorAfinidade", new LeafGoal("formarGrupoPorAfinidade", "String", ""));
   addGoal("formarGrupoPorPerfisAprendizagem", new Goal("formarGrupoPorPerfisAprendizagem",
       "String", ""));

   addRight("exibirDicasDeFormacao", new Right());
   addRight("criarGrupoPorPerfil", new Right());
   addRight("criarGrupoAfinidade", new Right());
   addRight("integrarGrupoAfinidade", new Right());
   addRight("integrarGrupoPerfil", new Right());
   addRight("requisitaCoordenadorAcao", new Right());

   initialize();
   }
}


import jamder.structural.*;
import jade.core.behaviours.Behaviour;
import jamder.Organization;
import jamder.agents.GenericAgent;
import jamder.roles.*;
import jamder.behavioural.*;

public class CoordenadorAgRole extends ProactiveAgentRole {
   //Constructor
   public CoordenadorAgRole (String name, Organization owner, GenericAgent player) {
   super(name, owner, player);
   addBelief("crencasCoordenador.pl", new Belief("crencasCoordenador.pl", "String", ""));

   addGoal("requisitarAcaoAgente", new LeafGoal("requisitarAcaoAgente", "String", ""));

   addRight("verificaAcaoDoAgente", new Right());
   addRight("requisitaAcao", new Right());

   initialize();
   }
}
```

Figura 38 Classes dos Papéis de Agente geradas.

6.3 Ambiente e Organização

Por fim, as classes de Organização e Ambiente deste SMA são exibidas através da Figura 39. A estrutura destas classes é mais simples, pois apenas contém as entidades que habitam nelas e a criação das instâncias que habitam nelas.

```
import jamder.Organization;
import jamder.roles.*;
import jamder.structural.*;
import jamder.behavioural.*;

public class MoodleOrg extends Organization {
   //Constructor
   public MoodleOrg (String name, Environment env, AgentRole agRole, Organization org) {
     super(name, env, agRole, org);

   }
}


import jamder.environment;
import jamder.Organization;
import jamder.roles.AgentRole;
import jamder.agents.GenericAgent;

public class MoodleEnv extends Environment {
  public MoodleEnv (String name, String host, String port) {
    super(name, host, port);
    Organization MoodleOrg = new Organization("MoodleOrg", this, null);
    addOrganization("MoodleOrg", MoodleOrg);

    GenericAgent AjudanteAg = new AjudanteAg("AjudanteAg", this, null);
    AgentRole AjudanteAgRole = new AgentRole("AjudanteAgRole", MoodelOrg, AjudanteAg);
    addAgent("AjudanteAg", AjudanteAg);

    GenericAgent BuscadorAg = new BuscadorAg("BuscadorAg", this, null);
    AgentRole BuscadorAgRole = new AgentRole("BuscadorAgRole", MoodelOrg, BuscadorAg);
    addAgent("BuscadorAg", BuscadorAg);

    GenericAgent CompanheiroAg = new CompanheiroAg("CompanheiroAg", this, null);
    AgentRole CompanheiroAgRole = new AgentRole("CompanheiroAgRole", MoodelOrg,
CompanheiroAg);
    addAgent("CompanheiroAg", CompanheiroAg);

    GenericAgent CoordenadorAg = new CoordenadorAg("CoordenadorAg", this, null);
    AgentRole CoordenadorAg = new AgentRole("CoordenadorAg", MoodelOrg, CoordenadorAg);
    addAgent("CoordenadorAg", CoordenadorAg);

    GenericAgent FormadorAg = new FormadorAg("FormadorAg", this, null);
    AgentRole FormadorAgRole = new AgentRole("FormadorAgRole", MoodelOrg, FormadorAg);
    addAgent("FormadorAg", FormadorAg);

    GenericAgent PedagogicoAg = new PedagogicoAg("PedagogicoAg", this, null);
    AgentRole PedagogicoAgRole = new AgentRole("PedagogicoAgRole", MoodelOrg, PedagogicoAg);
    addAgent("PedagogicoAg", PedagogicoAg);
  }

  // Additional attributes

  // Additional methods
}
```

Figura 39 Classes MoodleOrg e MoodleEnv geradas.

A transformação dos modelos em código através dos *templates* em Acceleo propôs a gerar apenas o esqueleto das classes e dos métodos das entidades e suas características. Algumas entidades necessitaram incluir, como um padrão de nomenclatura, um sufixo devido à possibilidade de existir inúmeras propriedades na entidade e com isto, organizar melhor sua estrutura, a saber: *Plan* (*Plan*), *Action* (*Ac*), *Goal* (*G*), *Belief* (*B*). Esta padronização de nomes exclui apenas as classes *Duty*, *Right* e *Protocol*, pois os mesmos, ao serem gerados, suas instâncias são criadas diretamente e não são referenciadas no mesmo método, sem a necessidade de uma variável. Esta nomenclatura adotada proporciona um melhor entendimento das características envolvidas, dando a possibilidade de serem utilizadas no construtor da entidade de uma forma mais fácil. Outra vantagem é que ajuda a identificar mais rápido que tipo de classe ou variável foi gerada e evita possíveis erros de código, melhorando a qualidade do código.

Neste estudo de caso, JAMDER foi aplicado no protótipo de um problema real, ilustrando a adequação da extensão de JADE para implementar entidades no contexto de MAS-ML 2.0. O código fonte de JAMDER e do estudo de caso completo, assim como os templates utilizados, podem ser obtidos através do site http://code.google.com/p/jamder/.

7 CONCLUSÃO

O adequado mapeamento entre modelagem e implementação tornam-se de fundamental importância para o desenvolvimento de um sistema, de forma a garantir consistência e rastreabilidade entre os artefatos de modelagem e o código. Neste contexto, o presente trabalho de dissertação objetivou o desenvolvimento de uma estratégia para a geração de código a partir de modelos, tendo como base os artefatos gerados a partir de ferramentas existentes de suporte a SMA. A definição da estratégia envolve o estabelecimento de propriedades e características com o objetivo de determinar os elementos de modelagem a serem contemplados na codificação.

Este mapeamento considera, além das entidades tipicamente encontradas em SMA, as diferentes arquiteturas internas de agentes, a saber: reativo simples, reativo baseado em conhecimento, pró-ativo baseado em objetivo guiado por busca, pró-ativo baseado em utilidade e pró-ativo baseado em objetivo guiado por plano. Desta forma, fez-se necessário um mapeamento entre modelagem e implementação, o qual originou uma extensão do *framework* JADE devido à ausência de algumas entidades de modelagem no mesmo. O conjunto de classes associadas ao JADE propostas neste trabalho recebeu o nome de JAMDER. JAMDER não contempla somente as diferentes arquiteturas internas, mas as entidades de modelagem, suas características e relacionamentos.

A utilização do framework JAMDER para outras linguagens de modelagem não é possível, pois algumas entidades são específicas para a linguagem MAS-ML 2.0, como por exemplo, os três tipos de papel de agente, enquanto que AUML [AUML, 2011] define um tipo de papel e AORML [AORML, 2011] não possui. Isto também se aplica para alguns agentes, pois em MAS-ML 2.0 são definidos cinco tipos, enquanto que AUML e AORML não oferecem suporte aos agentes reativos. Em relação à organização, é necessário que contenha também as mesmas características que MAS-ML 2.0, visto que em AUML e AORML não são definidas as propriedades de organização. Em relação a ambientes, exceto o MAS-ML, as outras linguagens de modelagem vistas neste trabalho não definem o conceito de ambiente.

O *plugin* Acceleo foi utilizado para dar suporte à geração de código através de *templates*, elaborados neste trabalho, para criar as classes em JAMDER que atendam à modelagem MAS-ML 2.0. Os *templates* foram organizados para cada entidade pertencente a MAS-ML, a saber: Agente, Organização, Ambiente, Papel de agente, Objeto e Papel de objeto. Um dos pontos positivos da geração do código é que proporciona a automatização da passagem da fase de projeto para a fase de implementação através da execução dos modelos, proposto pelo padrão de arquitetura orientada a modelos, MDA.

O resultado pretende beneficiar a equipe envolvida no desenvolvimento de SMAs ao ajustar a modelagem de sistemas e sua implementação diminuindo o tempo gasto que se levaria em mapear as informações e gerar código manualmente.

Um estudo de caso baseado em uma aplicação real (*Moodle*) foi utilizado para ilustrar o processo de geração de código a partir dos modelos de MAS-ML *Tool* e de mostrar a sua aplicabilidade e adequação para o desenvolvimento de SMA através do código JAMDER.

7.1 Trabalhos Futuros

Existem algumas sugestões de pontos que não puderam ser tratadas e, poderão ser desenvolvidas com o intuito de dar continuidade ao trabalho apresentado nesta dissertação. Dentre eles podem ser citados:

1. Devido à ferramenta MAS-ML *Tool* ainda se encontrar em desenvolvimento, algumas características intrínsecas ao complemento da geração do código não puderam ser realizadas, como a associação das ações ao plano predefinido. Mesmo não podendo ser gerado, os *templates* desenvolvidos possibilitam a inclusão de ações, porém no aprimoramento de MAS-ML *Tool* contendo as características que ainda faltam;
2. Cada *template* desenvolvido neste trabalho é executado individualmente, um para cada tipo de entidade. Desta forma, a ideia de criar uma cadeia de chamadas destes *templates* de forma sequencial possibilita uma redução do tempo para se gerar as classes também.
3. Todos os resultados das extensões realizadas, extensão da implementação e geração de código, estão direcionados a adaptar a ferramenta de

implementação JADE à linguagem de modelagem MAS-ML 2.0. Adaptações relativas a outras linguagens de implementação referenciadas neste trabalho para a mesma ferramenta de modelagem também podem ser criadas. De forma análoga à abordagem aqui proposta, se torna necessário fazer uma extensão das classes da linguagem de implementação escolhida de forma a contemplar as características necessárias previstas na modelagem e, criar novos *templates* direcionados à linguagem específica.

4. Para haver a possibilidade de incluir um código fonte dentro de um método definido no agente no momento de sua geração, é necessário adicionar um novo campo no compartimento *Action* da ferramenta MAS-ML *Tool*. O valor deste campo conteria o código fonte definido em tempo de modelagem, onde o desenvolvedor iria incorporar uma nota textual representando uma atividade do agente, e.g. rede neural ou lógica fuzzy. A fim de completar a geração, a estrutura do *template* Acceleo para agentes também precisaria ser alterada para contemplar esta adaptação.

REFERÊNCIAS BIBLIOGRÁFICAS

ACCELEO, **Acceleo OpenSource**, disponível em: http://www.acceleo.org/, acessado em 13 de outubro de 2011.

ANDROMDA, **AndroMDA**,disponível em: http://www.andromda.org/, acessado em 19 de setembro de 2011

AORML, **Agent-Object Relationship Modeling Language**, disponível em: https://oxygen.informatik.tu-cottbus.de/aor/, acessado em 05 de agosto de 2011.

AUML, **Agent Unified Modeling Language**, disponível em: http://www.auml.org/, acessado em: 15 de abril 2011

AUML2-TOOL, **AUML-2 & Interaction Diagrama Tool**. Disponível em http://waitaki.otago.ac.nz/~michael/auml. Acessado em 05 de dezembro de 2010.

BEYDEDA, S., BOOK M., e GRUHN, V. **Model-driven Software Development**. Birkhäuser, 2005.

BELLIFEMINE, F. L.; CAIRE, G.; GREENWOOD, D. **Developing Multi-Agent Systems with JADE**. [S.l.]: Wiley, 2007. (Wiley Series in Agent Technology).

BLOIS, M., LUCENA, C. **MULTI-AGENT SYSTEMS AND THE SEMANTIC WEB** - The SemanticCore Agent-Based Abstraction Layer. In: ICEIS - International Conference on Enterprise Information Systems, 2004, Porto. Proceedings of Sixth International Conference on Enterprise Information Systems ICEIS 2004. Porto: INSTICC, 2004. p. 263-270.

CASTRO, J.; ALENCAR, F.; SILVA, C. **Engenharia de Software Orientada a Agentes.** In: Karin Breitman; Ricardo Anido. (Org.). Atualizações em Informática. Rio de Janeiro: PUC-Rio, p. 245-282. 2006.

CHOREN, R. and LUCENA, C. **Agent-Oriented Modeling Using ANote**, 3rd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2004), 3rd; The Institution of Electrical Engineers, IEE, Stevenage, UK, 2004, pp. 74-80, ISBN: 0-86341-431-1, May 24-25. 2004

DE MARIA, B. A. **Usando a abordagem MDA no desenvolvimento de sistemas multi-agentes**. Dissertação de Mestrado – Pontífica Universidade Católica do Rio de Janeiro. 2004.

DÁRIO, C. F. B. **Uma Metodologia Unificada para o Desenvolvimento de Sistemas Orientados a Agentes**. Dissertação de Mestrado Universidade Estadual de Campinas. 2005.

ECLIPSE, **Eclipse Platform**. Disponível em: www.eclipse.com acessado em 12 de outubro de 2011.

FARIAS, K.; NUNES, I.; SILVA, V. T. da; LUCENA, C. J. P. de. **MAS-ML Tool**: Um Ambiente de Modelagem de Sistemas Multi-Agentes. Fifth Workshop on Software Engineering for Agent-oriented Systems (SEAS@SBES 09). 2009.

FAZZIKI, A. E., NOUZRI, S., SADGAL, M. **Towards an MDA Based Multi-agent Approach for Information System Development**. In International Journal of Digital Society (IJDS). 2010.

FIPA, **Foundations of Intelligent Physical Agents**, disponível em: http://jade.tilab.com/, acessado em: 13 de fevereiro de 2011.

FRANCE, R.; RUMPE, B. **Model-Driven Development of Complex Software: A Research Roadmap**, in Future of Software Engineering (FOSE'07) co-located with ICSE'07, Minnesota, EUA. Maio, 2007.

HUBER, M. J., **JAM Agent**. Disponível em: http://www.marcush.net/IRS, acessado em 14 de setembro de 2011.

HUGET ***apud*** SILVA, V. T. **Uma linguagem de modelagem para sistemas multi-agentes baseada em um framework conceitual para agentes e objetos**. Tese de doutorado. Rio de Janeiro: PUC, Departamento de Informática. 2004.

GONÇALVES, E. J. T. **Modelagem de arquiteturas internas de agentes de software utilizando a linguagem MAS-ML 2.0**. Dissertação de Mestrado. Universidade Estadual do Ceará. Centro de Ciência e Tecnologia. Fortaleza, 2009.

GONÇALVES, E. J. T.; GOMES, G. F.; CAMPOS, G. A. L. de; SOUZA, J. T.; CORTÉS, M. I.; FEITOSA, R. e LOPES, Y. S. **Combinando MAS-School, ANote e JADE para o Desenvolvimento de Sistemas Multi-agentes**. InfoBrasil 2010, Fortaleza, CE, Brasil, 2010a.

GONÇALVES, E. J. T., FARIAS, K., CORTES, M. I., FEITOSA, R. G. F., SILVA, V. T. Modelagem de Organizações de Agentes Inteligentes: uma Extensão da MAS-ML Tool. AutoSoft 2010. Salvador. 2010b.

GONÇALVES, E. J. T., FARIAS, K., CORTES, M. I., FEIJO, A. R., OLIVEIRA, F. R., SILVA, V. T. **MAS-ML TOOL**: A Modeling Environment For Multi-Agent Systems. ICEIS 2011. Pequim. 2011.

JACK. **JACK Agent Language**. Disponível em: http://www.agent-software.com.au/products/jack/, acessado em: 23 de fevereiro de 2011.

JADE, **Java Agent Development Framework**, disponível em: http://jade.tilab.com/, acessado em: 15 de fevereiro de 2011.

JADEX, **JADE XML**, disponível em: http://JADEX-agents.informatik.uni-hamburg.de/, acessado em: 16 de fevereiro de 2011.

JASON, **Java-based interpreter for an extended version of AgentSpeak**. [S.l.], disponível em: http://jason.sourceforge.net/, acessado em: 27 de fevereiro de 2011.

JENNINGS, N. R. **Agent-Oriented Software Engineering**, in F. J. Garijo & M. Boman, eds, 'Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a multi-agent World: Multi-Agent System Engineering (MAAMAW-99)', Vol. 1647, Springer-Verlag: Heidelberg, Germany, pp. 1-7. 1999.

LOPES, Y. S.; GONÇALVES, E. J. T.; CORTÉS, M. I. e FREIRE, E. S. S. **JAMDER: Uma Extensão do Framework JADE com Foco em Agentes**. II Congresso Brasileiro de Software: Teoria e Prática (CBSoft 2011) no 2º Autonomous Software Systems (AutoSoft 2011), São Paulo, 2011a.

LOPES, Y. S.; GONÇALVES, E. J. T.; CORTÉS, M. I. e FREIRE, E. S. S. **Extending JADE Framework to Support Different Internal Architectures of Agents**. 9th European Workshop on Multi-agent Systems (EUMAS 2011), Maastricht, Holanda, 2011b.

LUCAS, D. C. **QCodeGenerator**: um gerador de código multilinguagem baseado em templates. Faculdade Cenecista Nossa Senhora dos Anjos, Gravataí, Rio Grande do Sul, 2000.

MOODLE, **Moodle**. Disponível em: http://moodle.org, acessado em: 10 de novembro de 2011.

NUNES, I. O. **Implementação do Modelo e da Arquitetura BDI**. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. Rio de Janeiro, 2007.

ODELL, J. D., PARUNNAK, H. V. D. and BAUER B. S. V. **Representing Agent Interaction Protocols in UML**. 22nd International Conference on Software Engineering (ICSE), pp. 121–140, 2001

OMG. Object Management Group. Disponível em: http://www.omg.org, acessado em: 12 de dezembro de 2011.

PADILHA, T. P. P., JÁCOME, T.F. **O Uso de Técnicas de Modelagem de Agentes em Ambientes Educacionais**. In: VI Congresso Iberoamericano de Informática Educativa. 2002.

PADGHAM L., THANGARAJAH, J. and WINIKOFF, M. **Prometheus Design Tool**, in 23th AAAI Conference on Artificial Intelligence, Chicago, EUA, pp.1882-1883 2008.

RAO, A. S. and GEORGEFF, M. P. **BDI-agents**: From Theory to Practice , In Proceedings of the First International Conference on Multiagent Systems (ICMAS'95), San Francisco, 1995.

RUSSELL, S. NORVIG, P. **Inteligência Artificial**: uma abordagem moderna. 2 Ed. São Paulo: Prentice-Hall. 2004.

SANTOS, D. R. **Um Metamodelo para a Representação Interna de Agentes de Software**. Dissertação de Mestrado. Porto Alegre: PUC. 2008.

SILVA, V.; GARCIA, A.; BRANDAO, A.; CHAVEZ, C.; LUCENA, C.; ALENCAR, P. **Taming Agents and Objects in Software Engineering** In: Garcia, A.; Lucena, C.; Zamboneli, F.; Omicini, A; Castro, J. (Eds.), Software Engineering for Large-Scale Multi-Agent Systems, Springer-Verlag, LNCS 2603, pp. 1-26, 2003, ISBN 978-3-540-08772-4. 2003.

SILVA, V. T. **Uma linguagem de modelagem para sistemas multi-agentes baseada em um framework conceitual para agentes e objetos**. Tese de doutorado. Rio de Janeiro: PUC, Departamento de Informática. 2004.

SILVA, V., Lucena, C., **From a Conceptual Framework for Agents and Objects to a Multi-Agent System Modeling Language**. Journal of Autonomous Agents and Multi-Agent Systems 9(1-2), Kluwer Academic Publishers, p. 145-189. 2004.

SILVA, V. T. da; CHOREN, R.; LUCENA, C. J. P. **Using UML 2.0 Activity Diagram to Model Agent Plans and Actions**. The International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'2005), 4th. Proceedings of the International Conference on Autonomous Agents and

Multi-Agent Systems, Netherlands, Holanda, pp. 594-600, v. 2, n.1, ACM, ISBN: 1-59593-094-9. 2005.

SILVA, V. T. da; CHOREN, R.; LUCENA, C. J. P. **MAS-ML**: A Multi-Agent System Modeling Language. Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA); In: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications; Anaheim, CA, USA, ISBN:1-58113-751-6, ACM Press, pp. 304-305 (Poster Session). 2007.

SOMMERVILLE, I. **Engenharia de Software**. Ed. Addison Wesley. São Paulo, 2007.

SOUZA, G. P. **Modelagem de Sistemas Distribuídos usando MDA**. Disponível em: http://saloon.inf.ufrgs.br/twiki-data/Disciplinas/CMP157 /TF09GiselePSouza/ TF09_2_RelatorioGiselePSouza.pdf, acessado em: 03 de dezembro de 2011.

VELOCITY. **Apache Velocity Project**. Disponível em: http://velocity.apache.org/, acessado em 03 de janeiro de 2012.

WAGNER, G. **The Agent-Object-Relationship Meta-Model**: Towards a Unified View of State and Behavior. Information Systems, v. 28, n.5, pp. 475–504. 2003.

WEISS, G. Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. MIT Press, Massachusetts.1999.

WOOLDRIDGE, M. **An Introduction to MultiAgent Systems**. Chichester: John Wiley & Sons Ltd., 2002. 348 p.

ZAMBONELLI, F.; JENNINGS, N.; WOOLDRIDGE, M. **Organizational abstractions for the analysis and design of multi-agent systems**. Em: Ciancarini, P.; Wooldridge, M. (Eds.) Agent-Oriented Software Engineering, LNCS 1957, Berlin: Springer, p. 127. 2001.

APÊNDICE I : COMPARATIVO DAS FERRAMENTAS DE MODELAGEM

As seguintes ferramentas de modelagem foram analisadas levando em consideração se possuem suporte a ambiente e papéis de agente, diferentes arquiteturas internas, inclusive as definidas por [RUSSELL; NORVIG, 2004], se possuem uma ferramenta de modelagem, se estendem UML e se geram código. A análise considerou estes fatores e o resultado obtido segue de acordo com a Tabela 5:

Tabela 5 Tabela comparativa das linguagens de modelagem

Características	**Modelagens**			
	AUML	**AORML**	**ANote**	**MAS-ML 2.0**
Possui Ambientes	Não	Não	Não	Sim
Arquiteturas internas de Agentes	Não oferece suporte a agentes reativos	Não oferece suporte a agentes reativos	Apenas agentes Baseados em objetivos guiados por planos	Sim
Possui Papéis	Sim	Não	Não	Sim
Ferramenta Sup.	AUML2 *Tool* (protótipo)	Não possui	Albatroz	MAS-ML *Tool*
Gera Código	Não	Não	ASYNC (mas não suporta ontologia)	Não
Estende UML	Sim	Sim	Não	Sim

Os seguintes motivos levaram a esta escolha de MAS-ML 2.0:

- Extensão conservativa de UML;
- Possuir um framework conceitual;
- Oferecer suporte a objetos convencionais;
- Identificar papéis;
- Modelar adequadamente ambientes e a interação entre agentes e ambientes;
- Possuir a ferramenta de suporte a modelagem MAS-ML *Tool*. Na sua versão atual a ferramenta dá suporte a três diagramas estáticos: classes, organização e papéis (em desenvolvimento).