Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Confiabilidade em Sistemas Multiagentes

João Paulo de Freitas Matos

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Célia Ralha

Brasília
2012

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Marcus Vinícus Lamar

Banca examinadora composta por:

Prof. Célia Ralha (Orientador) — CIC/UnB
Prof. Genaína Nunes — CIC/UnB
Prof. Professor II — CIC/UnB

CIP — Catalogação Internacional na Publicação

Matos, João Paulo de Freitas.

Confiabilidade em Sistemas Multiagentes / João Paulo de Freitas Matos. Brasília : UnB, 2012.
91 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2012.

1. Confiabilidade, 2. Sistemas Multiagentes

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Confiabilidade em Sistemas Multiagentes

João Paulo de Freitas Matos

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Célia Ralha (Orientador)
CIC/UnB

Prof. Genaína Nunes
CIC/UnB

Prof. Professor II
CIC/UnB

Prof. Marcus Vinícus Lamar
Coordenador do Bacharelado em Ciência da Computação

Brasília, 12 de dezembro de 2012

Dedicatória

Dedico a....

Agradecimentos

Agradeço a....

Abstract

A ciência...

Palavras-chave: Confiabilidade, Sistemas Multiagentes

Abstract

The science...

Keywords: Reliability, Multiagent systems

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Com o crescente desenvolvimento da computação que penetra cada vez mais em diversas áreas de conhecimento, a demanda por processamento tende a crescer rapidamente, tornando o desenvolvimento de aplicações cada vez mais complexo, exigindo cada vez mais desempenho e consequentemente poder de processamento. Para a execução dessas aplicações em tempo hábil, são necessários investimentos cada vez mais altos em *hardwares* melhores, existindo porém um fator limitante (custo ou tecnologia).

Além disso, atualmente a grande quantidade de informação disponível exige análises cada vez mais precisas e detalhadas da informação processada tendo em vista a ajuda de tomada de decisões. Dessa forma, aplicações que antes eram centralizadas em uma única máquina transformaram-se em aplicações distribuídas em várias máquinas que são concorrentes e assíncronas.

A partir dessa motivação, aplicações são projetadas para rodar de forma descentralizada, com componentes e serviços rodando em diversos lugares distintos e comunicando-se uma com as outras através de mensagens. Cada módulo pode ter um objetivo específico, como capturar e processar eventos no ambiente computacional, processar informações, persistir eventos no banco de dados, enfim, uma vasta gama de operações que são assíncronas e independentes. A arquitetura dessas aplicações é projetada objetivando o alto paralelismo, flexibilidade, interoperabilidade, dentre outros aspectos.

Diversos *frameworks* tentam lidar com o problema da computação distribuída, porém alguns usam arquiteturas que são baseadas em processamento ordenado: Os dados são processados ordenadamente, não usufruindo de todo o processamento que poderia ocorrer se fosse realmente paralelo. Outros *frameworks* são embasados em tecnologias que não são recomendáveis em um ambiente distribuído: O uso de recurso um compartilhado e blocante, que pode prejudicar o processamento em larga escola.

A computação distribuída toma formas ainda mais interessantes quando aplicada a contextos sensíveis a sociedade em geral. Áreas de atuação como Bolsa de Valores, Análise de Redes Sociais, Análise de Mídia Social, Informática na Educação, dentre outras. Em especial a esta última área, a possibilidade de auxílio no aprendizado do aluno eleva a importância deste setor na computação.

Na perspectiva da Informática na Educação (IE), a abordagem chamada Sistemas Tutores Inteligentes permite a representação de conhecimento de forma muito interessante. É possível a construção um modelo onde se representa o estudante (o objeto a quem se deve ensinar), o domínio do conteúdo (o conteúdo a ser ensinado) e o modelo peda-

gógico (a forma a ser ensinada). Esta abordagem permite o uso de vários conceitos da Inteligência Artificial para determinação de estilos de aprendizagem, dentre outras possibilidades. Dessa forma, aplicações tendem a ser bastante complexas vistas ao alto grau de processamento que este ambiente pode assumir.

1.1 Problema

Os ambientes educacionais de aprendizagem não possuem inferência de modelo do estudante, pois sua implementação seguindo modelo cliente-servidor não é apropriada para tal finalidade. Além disso, a alta carga da aplicação devido ao fato de muitos acessos dos alunos pode prejudicar uma aplicaçao centralizada, necessitando de um sistema que processasse em múltiplos locais.

1.2 Objetivos do Projeto

Tendo em vista o cenário atual apresentado, o objetivo geral deste trabalho é propor a arquitetura distribuída de um ambiente computacional, bem como sua construção, baseada em Sistemas Multiagentes, capaz de construir e manter um modelo do estudante, a partir do qual os estilos de aprendizagens desse estudante poderão ser identificados e informados ao docente por meio de uma interface web.

Os objetivos específicos serão devidamente justificados na próxima subseção, bem como idealizada a forma de atingi-los.

Usando tecnologias existentes e consolidadas, a arquitetura proposta irá usar o framework *JADE*, que é completamente desenvolvido na linguagem *JAVA* e simplifica a implementação de Sistemas Multiagentes (SMA) que cumprem as especificações FIPA. A arquitetura proposta também englobará uma interface web que utiliza a plataforma *open source JBoss Seam*, desenvolvida para auxiliar a construção de aplicações dinâmicas para a internet de forma simples e ágil.

1.3 Objetivos Específicos do Projeto

Este trabalho foi norteado pelo documento do projeto Frank [10], sendo construído como uma proposta ao problema nele proposto. A partir deste documento, é possível identificar os objetivos específicos derivados da construção do ambiente computacional para manutenção do modelo do aluno. Mais especificamente, os objetivos específicos deste trabalho são:

- Objetivo específico 1: Obter uma modelagem da arquitetura geral do SMA Frank utilizando-se da metodologia *Multiagent System Engineering* (MASE), proposta como uma solução de Engenharia de Software para o desenvolvimento de SMA;
- Objetivo específico 2: Obter uma implementação da arquitetura geral do SMA Frank;
- Objetivo específico 3: Obter uma implementação da arquitetura do agente assistente de cognição;

- Objetivo específico 4: propor uma interface do agente assistente de cognição com o estudante;
- Objetivo específico 5: propor uma interface do agente assistente de cognição com o docente;

A seguinte estrutura desse trabalho consiste na divisão de capítulos visando facilitar a leitura e organizar os conceitos que perfazem o desenvolvimento deste trabalho:

- O presente capítulo contém a introdução, onde o trabalho é justificado e os objetivos são esclarecidos.
- Capítulo 2 contém todos os fundamentos teóricos necessários para o desenvolvimento desse trabalho.
- Capítulo 3 contém a proposta de solução composta pela metodologia, modelagem da arquitetura e implementação.
- Capítulo 4 contém a experimentação e análise de resultados.
- Por fim, o capítulo 5 relata a conclusão e trabalhos futuros.

Capítulo 2

Fundamentos básicos

Este capitulo apresenta os principais conceitos e definições necessários para o entendimento deste trabalho. A seção 2.1 apresenta alguns conceitos básicos em *Unified Modeling Language* (UML), que são necessários para o entendimento da modelagem deste trabalho. A seção 2.2 disserta sobre conceitos a respeito da informática na educação. A seção 2.3 aborda a teoria sobre Sistemas Multiagentes necessária para este trabalho. A seção 2.4 contém a metodologia *Multiagent System Engineering*, desenvolvida para a criação de Sistemas Multiagentes A seção 2.5 e 2.6 abordam o funcionamento dos frameworks JADE e Jboss Seam, respectivamente. Por fim, a seção 2.7 detalha alguns trabalhos correlatos.

2.1 Unified Modeling Language

A Linguagem Unificada de Modelagem, *Unified Modeling Language* (UML) é uma linguagem visual que foi desenvolvida para a representação do software por meio de imagens, objetivando o entendimento dos artefatos de forma rápida e clara e resultando em uma semântica para o projeto em questão. De acordo com [12] o UML faz parte de uma família de notações gráficas que ajudam na descrição e concepção de sistemas de software, principalmente em sistemas concebidos utilizando o paradigma da orientação à objetos (OO).

O UML é um padrão não proprietário, controlado pelo consórcio *Object Management Group* [3]. Seu nascimento é datado em 1997 [12], surgindo a partir da união de diversas linguagens e ferramentas de surgiram na década de 80 e 90. A linguagem ajuda o entendimento de como o software foi projetado, como ocorre a comunicação entre seus objetos, como suas classes são organizadas, quais são os atores que são envolvidos na utilização do software, dentre outras possibilidades de representação.

Em [12], é possível separar o uso do UML de três formas distintas, diferindo entre as modelagens utilizadas e o objetivo de uso. As três formas são: Rascunho, planta de software e como linguagem de programação.

A utilização como rascunho é utilizada para facilitar a comunicação entre as pessoas envolvidas no projeto, sejam desenvolvedores discutindo funcionalidades do software ou gestores explicando funcionalidades em alto nível. O objetivo neste uso é a comunicação de alguns aspectos do sistema de forma rápida, sem a necessidade de formalizar artefatos para o projeto.

A utilização do UML como planta de software são documentos detalhados que são criados para documentação do software, sendo divida em duas sub-categorias: Engenharia reversa e engenharia avante. Na engenharia reversa, os diagramas são gerados a partir de uma ferramenta que faz a leitura do código fonte e gera os diagramas desejados, que são utilizados para auxiliar o leitor no entendimento do sistema. Na engenharia avante, a idéia é modelar o sistema detalhadamente antes de qualquer desenvolvimento, prevendo quais serão os módulos do sistema, bem como a sua a comunicação.

No uso como linguagem de programação, o UML é utilizado para geração de código executável por ferramentas avançadas de modelagem. Esse modo requer a modelagem de estado e comportamento do sistema, para fins de detalhar todo o comportamento e lógica do sistema em código.

2.1.1 Diagramas UML

O UML 2 descreve 13 tipos de diagramas que podem ser categorizados por estruturais e comportamentais. A imagem 2.1 ilustra essa categorização de diagramas.

Apesar da grande quantidade diagramas envolvidos no UML, nem todos os processos de desenvolvimentos de software utilizam todos eles. A maioria das pessoas utiliza-se de um conjunto de poucos diagramas para modelagem do sistema. Nas próximas subseções, serão detalhados apenas os seguintes diagramas, que são necessários para o entendimento deste trabalho:

- Diagrama de Caso de uso
- Diagrama de sequência

Diagrama de Caso de Uso

Casos de uso são relatos textuais que são utilizados para descobrir e descrever os requisitos do sistema. Consiste da descrição de como um ator utiliza uma funcionalidade do sistema para atingir algum objetivo relacionado. De acordo com [14], os casos de uso devem ser prioritariamente desenvolvidos de forma textual e o seu respectivo diagrama deve ser desenvolvido de forma secundária, somente para ilustrar o relato textual.

Um dos objetivos do caso de uso é a facilidade do levantamento dos requisitos, tanto para os analistas de um sistema, quanto para os clientes envolvidos. A definição de uma modelagem em comum facilidade entre as partes faz do caso de uso uma boa maneira de simplificar o entendimento do comportamento do sistema [14], bem como envolver todos as partes interessadas do sistema(*stakeholders*) na construção do mesmo. De acordo com [9], o caso de uso é um contrato de como será o comportamento do sistema. Este contrato será feito por meio dos atores que existirão, da sua interação com o sistema, bem como os cenários existentes.

Duas definições fazem-se necessárias para o entendimento do caso de uso. A primeira delas é o "Ator"do caso de uso. Ele é um objeto com um comportamento definido no sistema. É possível definir o ator como uma pessoa, organização ou mesmo o próprio sistema (quando utiliza serviços do próprio sistema), desde que tenham sempre um papel relacionado. Existem três tipos de atores relacionados ao sistema:

- Ator Principal: Seus objetivos são satisfeitos por meio da utilização do sistema.

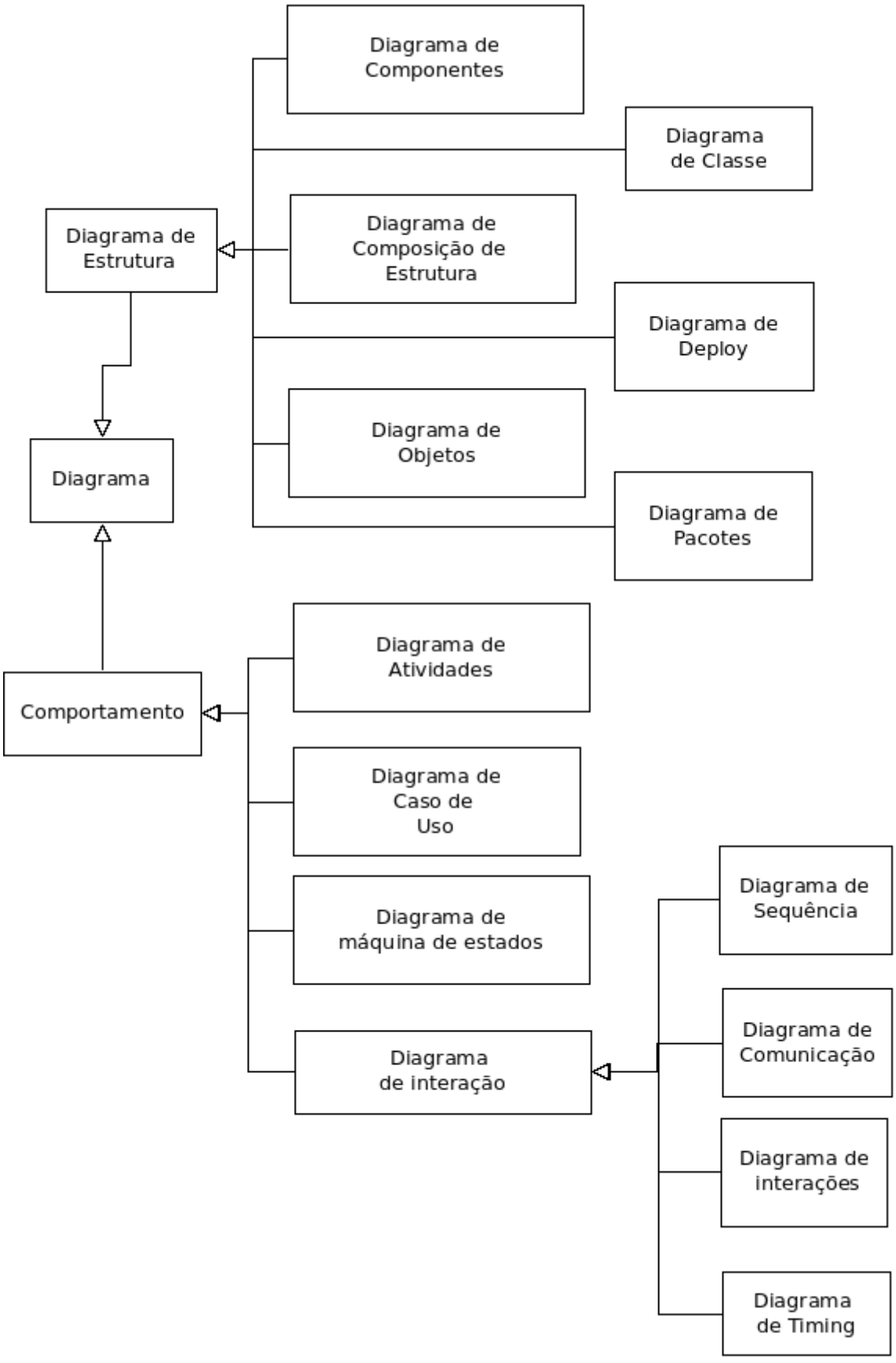

Figura 2.1: Diagramas categorizados por Estrutura e Comportamento. Fonte [12].

Tabela 2.1: Estruturação Detalhada de Caso de Uso

Seção do Caso de Uso	**Significado**
Nome do Caso de Uso	Nome do caso de uso, iniciando-se com um verbo
Escopo	Escopo descrito pelo caso de uso
Nível	Podem ser níveis de objetivo de usuário (quando descrevem os cenários para atingir o objetivo do usuário) ou nível de subfunção (subpassos para dar suporte a um objetivo de usuário)
Ator Principal	O ator que procura os serviços para atingir seus objetivos
Interessados e Interesses	Significado
Pré-Condições	Condições que antecedem o caso de uso e são necessárias para atingir os objetivos
Garantia de Sucesso	Objetos que podem ser analisados após a execução do caso de uso a fim de validar a correta execução do sistema
Cenário de Sucesso Principal	Chamado também de fluxo básico, este cenário descreve o fluxo principal do sistema que satisfaz os interesses dos interessados.
Extensões	Chamado também de fluxos alternativos, são fluxos auxiliares ou cenários de erros que são relacionados ao cenário de sucesso principal
Requisitos Especiais	Registram requisitos não funcionais do sistema e que estão relacionados com o caso de uso
Lista de Variantes Tecnologias de Dados	Listagem de dificuldades técnicas, desafios técnicos que valem a pena registrar no caso de uso
Frequência de ocorrência	Frequência de ocorrência deste caso de uso

- Ator de Suporte: Fornece algum serviço para o sistema.
- Ator de Bastidor: Expressa algum interesse pelo comportamento do caso de uso.

A segunda definição envolvida é a de cenário. Um cenário é uma sequência de interações entre os atores e o sistema. Os cenários são separados por ações de interesses de atores. Logo o caso de uso pode ser considerado como um conjunto de cenários de interações de atores com o sistema.

Dessa forma, o caso de uso deve deixar claro os requisitos funcionais do sistema, bem como o seu comportamento. Existem três formas de se escrever um caso de uso, diferindo em seu nível de formalidade e formatos: Resumido, informal e completo. Este trabalho usará o nível de caso de uso completo, devido ao fato de ser estruturado e mostrar mais detalhes. Os casos de uso são estruturados de diversas formas de acordo com a literatura, sendo a mais famosa [14] a estrutura utilizada por Alistar Cockburn [9] e detalhada na tabela 2.1

A diagramação do caso de uso por UML é uma forma de representação do sistema, mostrando fronteiras do sistema, comunicação e comportamento entre os atores. A figura 2.2

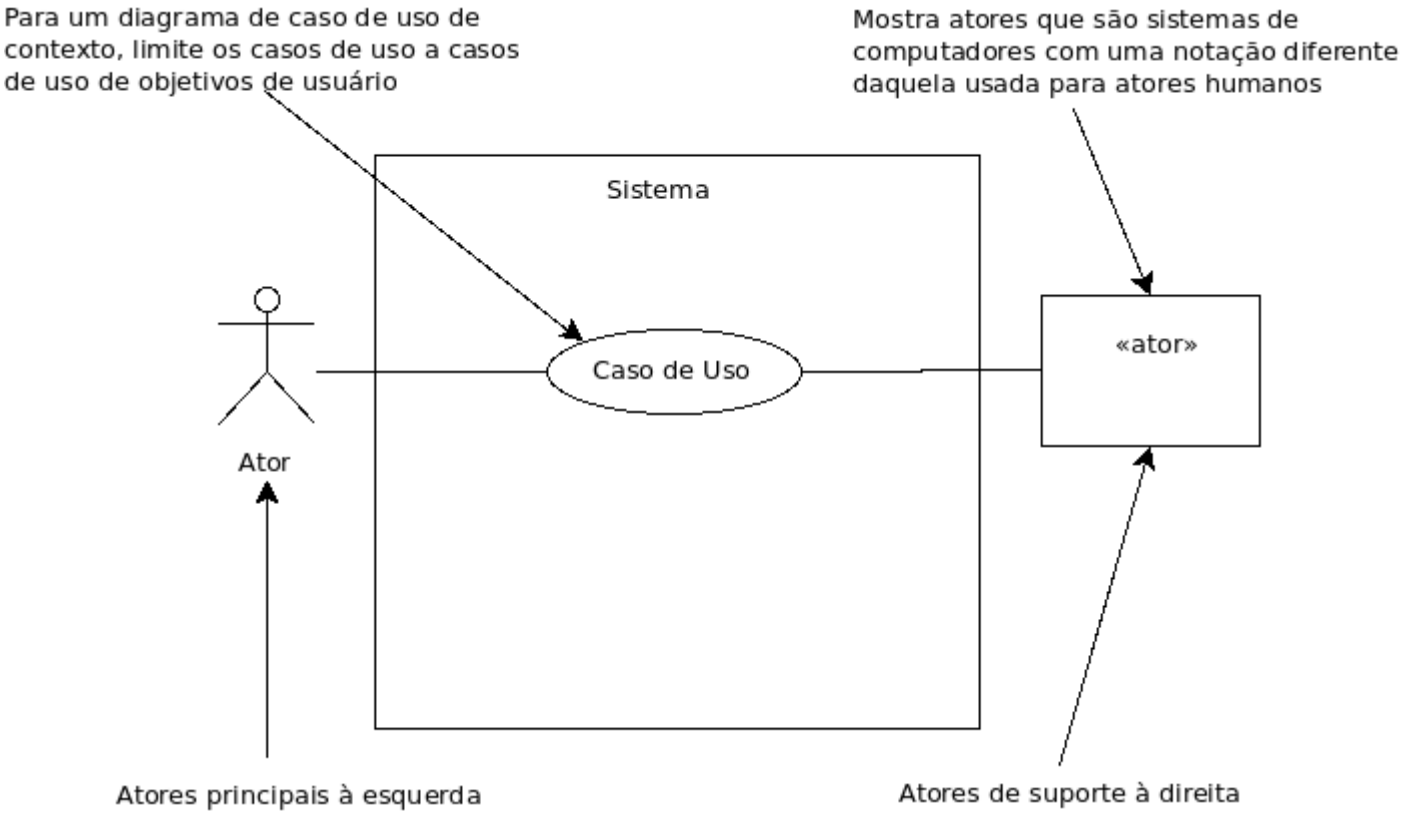

Figura 2.2: Sugestões de notação de caso de uso proposto por [14]

mostra a sugestão de diagramação do caso de uso, propondo forma de representação de atores, casos de uso e atores auxiliares.

Diagrama de Sequência

O Diagrama de Sequência é um documento criado para ilustrar os eventos descritos em um caso de uso de forma sequencial e temporal, mostrando a interação de atores externos ao sistema e os eventos que eles geram durante essa interação. A UML possui uma notação específica para este diagrama, onde todos os elementos são representados.

Neste diagrama, são representados para cada cenário do caso de uso os eventos que os atores geram, bem como a ordem da sua interação. No diagrama os atores são representados na parte superior (com a mesma notação do diagrama de caso de uso). Abaixo dos atores é apresentada a linha de tempo, crescendo de cima para baixo.

Durante a interação do ator com o sistema, eventos de sistema são gerados e iniciam toda a execução do cenário do caso de uso, ou operação do sistema. A execução dos eventos ocorre até o último evento cronológico na linha do tempo. Os eventos gerados pela interação entre os atores ocorrem na linha do tempo de forma cronológica e ordenada com a mesma ordem dos eventos do cenário do caso de uso.

A nomenclatura dos eventos deve sempre iniciar com um verbo, podendo ser seguida de um substantivo. Além disso, deve-se sempre expressar a nomenclatura em níveis genéricos verbais, nunca detalhando a funcionalidade do sistema.

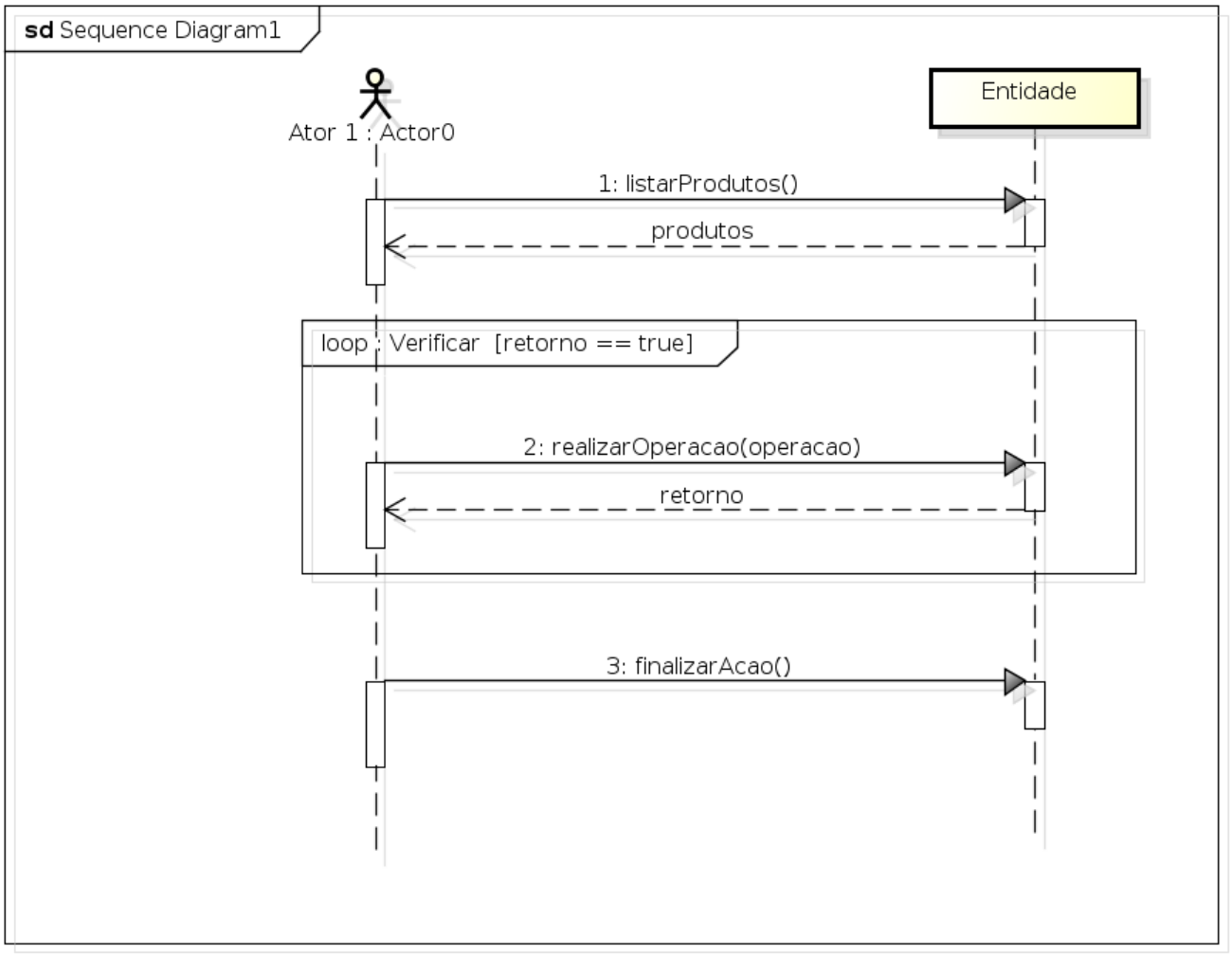

Figura 2.3: Notação de diagrama de sequência, exibindo a comunicação de um ator e uma entidade (sistema)

A imagem 2.3 ilustra a notação de diagramas de sequência com todos os pontos que foram até agora expostos. Nela, é possível identificar a interação entre um ator e uma entidade do sistema. É fácil identificar que os eventos estão ocorrendo de forma ordenada de cima para baixo.

O primeiro evento é iniciado pelo ator, onde o método (listarProdutos()) da Entidade é invocado. Esse método gera a resposta (produtos) para o ator. A interação seguinte acontece dentro de um retângulo, do tipo *loop* e de nome "Verificar". Esse retângulo permite que o diagrama de sequências represente um loop, onde todos os eventos serão repetidos enquanto a condição de guarda for verdadeira, no caso do exemplo, enquanto retorno for igual a *true*. O UML permite diversos operadores além do loop, como por exemplo a negação, a assertiva, o *break*, dentre outros.

Dentro do retângulo, o ator chama o método (realizarOperacao), enviado o parâmetro *operação*. A entidade retorna um valor, que será testado na guarda para a continuidade da conversação. Por fim, o ator chama o método *finalizarAcao* da entidade. Por não haver outra interação em seguida, o cenário é considerado como encerrado.

A importância do desenho de um diagrama de sequência está no fato de detalhar os eventos do sistema que são gerados pela interação de atores externos, pois assim é possível ter uma análise comportamental do sistema com base nesses eventos. Neste nível de análise, é possível estudar e projetar o comportamento do sistema no nível de projetar o que ele faz, porém sem necessariamente explicar como o faz [14].

O diagrama de sequência geralmente está relacionado com o caso de uso, primeiramente pelo fato de descrever um cenário de caso de uso, mas também pelo fato de o caso de uso conter todos os detalhes do cenário. Ele apenas deixará claro a interação entre os atores e os eventos derivados dessa interação.

2.2 Informática na Educação

2.3 Sistemas Multiagentes

Este capítulo visa introduzir o conceito de sistemas multiagentes (SMA). Para tanto é necessário mostrar conceitos que são base para o entendimento de SMA, iniciando pela apresentanção alguns conceitos a cerca de Inteligencia Artificial (IA). Em seguida o trabalho disserta sobre a teoria relacionada à Agente, bem como suas arquiteturas. Só então são apresentados os conceitos de Sistemas Multiagentes (SMA), comunicação em um ambiente SMA e por fim a teoria sobre ontologias.

2.3.1 Inteligência artificial

De acordo com [15] identificamos que a definição de IA pode variar em duas dimensões principais. Usando a definição de sistemas computacionais que agem racionalmente temos:

> *Computational Intelligence is the study of the design of intelligent agents.*

Nessa definição, é importante ressaltar que o agente é uma entidade que atua racionalmente, esperando-se que essa racionalidade e outras características o diferencie de simples programas.

Com o crescimento dos estudos relacionados a este campo a inteligência artificial ganhou várias áreas de atuação, permitindo assim a resolução de diversos desafios relacionados à aplicações modernas. Uma das áreas de atuação é o auxílio na execução de aplicações que resolvem problemas de alta complexidade.

Atualmente várias aplicações podem exigir demais de *hardwares* mais modestos tornando inviável o tempo de execução daquela aplicação, sendo necessário um investimento maior e consequentemente encarecendo o seu valor. Dessa forma outras abordagems fazem-se necessárias, como a distribução da aplicação em vários computadores que dividem a sua execução. Este é o campo de estudo da Inteligência Artificial Distribuída: Sistemas que são compostos por vários agentes coletivos, ou seja, distribuem o trabalho uns com os outros. Cada agente pode possuir uma capacidade diferente, sendo possível realizar a tarefa de modo paralelo.

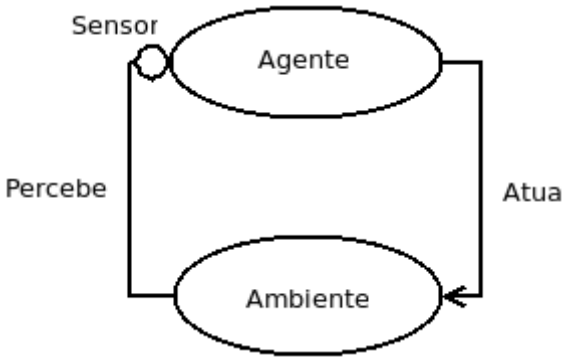

Figura 2.4: Esquematização do funcionamento básico de um agente em um ambiente.

2.3.2 Agente

De acordo com [17], agentes são entidades (reais ou virtuais) que funcionam de forma autônoma em um ambiente, ou seja, não necessitam de intervenção humana para realizar processamento. Esse ambiente de funcionamento do agente geralmente contém vários outros agentes e é possível a comunicação entre eles através do ambiente por meio de troca de mensagens.

Em geral o funcionamento de agentes acontece de forma a perceberem o ambiente em que estão por meio de sensores, fazem análises com base nessa interação inicial e por fim podem agir sobre o ambiente de forma a modifica-lo por meio de efetuadores. A figura 2.4 é uma ilustração do que foi dito.

Alguns agentes seguem o princípio de racionalidade básico: sempre objetivam suas ações pela escolha da melhor ação possível segundo seus conhecimentos. Logo é possível inferir que a ação de um agente nem sempre alcança o máximo desempenho, sendo desempenho o parâmetro definido para medir o grau de sucesso da ação de um agente com base nos seus objetivos. São estes os chamados Agentes Racionais.

Como dito anteriormente, agentes estão presentes em um ambiente. O agente não tem controle total do ambiente, ele pode no máximo influenciá-lo com a sua atuação ou criar outros agentes. Podemos separar ambientes em classes: Software, Físico e Relidade virtual (simulação de ambientes reais em software). De acordo com [22] temos, em geral, ambientes com propriedades inerentes à seu funcionamento:

- Observável: Neste tipo de ambiente, os sensores dos agentes conseguem ter percepção completa do ambiente. Por exemplo, um sensor de movimento consegue ter visão total em um ambiente aberto.
- Determinística: O próximo estado do ambiente é sempre conhecido dado o estado atual do ambiente e as ações dos agentes. O oposto do ambiente determinístico é o estocástico, quando não temos certeza do estado do ambiente. Por exemplo, agentes dependentes de eventos climáticos.
- Episódico: A experiência do agente é dividida em episódios, onde cada episódio é a percepção do agente e a sua ação.
- Sequêncial: A ação tomada pelo agente pode afetar o estado do ambiente e ocasionar na mudança de estado

Tabela 2.2: Listagem de sistemas multiagentes com propriedades de medida de performance, ambiente, atuadores e sensores

Tipo de agente	**Medida de performance**	**Ambiente**	**Atuadores**	**Sensores**
Sensores de estacionamento	Avarias no veículo	Carro e garagens	Freio do carro, controle de velocidade	Sensor de proximidade
Jogos com oponente computador	Quantidade de vitórias	Software	Realizar jogada	Percepção do tabuleiro
Agentes hospitalares	Saúde do paciente	Paciente, ambiente médico	Diagnósticos	Entrada de sintomas do paciente

- Estático: O ambiente não é alterado enquanto um agente escolhe uma ação.
- Discreto: Existe um número definido de ações e percepções do agente para o ambiente em cada turno.
- Contínuo: As percepções e ações de um agente modificam-se em um espectro contínuo de valores. Por exemplo, temperatura de um sensor muda de forma contínua.

Na tabela 2.2 mostramos alguns exemplos de agentes, apresentando as suas características já discutidas nesse trabalho.

A primeira linha da tabela 2.2 apresenta um exemplo de agente atuando em um veículo como um sensor de estacionamento. Responsável por auxiliar o motorista no ato de estacionar o carro, o seu ambiente é da classe físico (considerando o carro e o ambiente onde está o carro). Seu sensor de proximidade é a percepção do ambiente e, caso detecte que está próximo de um obstáculo, pode atuar nos freios dos carros diminuindo a velocidade e evitando colisões. Avarias no carro podem indicar um mal funcionamento do sensor.

A segunda linha da tabela é apresentado o exemplo de agente atuando em um jogo avulso. Esse ambiente é dito dinâmico, pois a cada jogada de um oponente (real ou não), o agente deve analisar a jogada feita pelo seu oponente, irá calcular sua próxima jogada e a realizará. O objetivo principal do agente é a vitória. O ambiente que o agente atua é um software e o seu atuador é um algum mecanismo que permite que ele realize a jogada. O sensor é o mecanismo no qual o agente irá perceber a jogada realizada pelo oponente.

Por fim, a última linha da tabela 2.2 expõe um exemplo de um agente médico atuando em um ambiente estático: Um paciente. Esse ambiente é classificado como estático por não ser alterado pelo agente nesse exemplo, porém é possível ser diferente em outras situações. O objetivo principal é monitorar a saúde do paciente, logo a medida de performance será a aproximação ou não do diagnóstico médico. Seu atuador não será diretamente no ambiente (corpo humano), será na forma de relatórios médicos e seus sensores podem variar de acordo com a doença a ser monitorada.

Conforme podemos encontrar em [22], podemos definir algumas noções gerais de agentes. A primeira, chamada de noção fraca, contém a maior parte dos agentes. Ela compreende os aspectos de *reatividade*, *proatividade* e *habilidade social*. O conceito de reatividade

está ligado com o agente perceber o ambiente e reagir. Proatividade é a característica do agente tomar a iniciativa e agir sem a necessidade de nenhum estímulo. Habilidade social é a capacidade de interação com outros agentes.

Já a noção forte de agente envolve os seguintes aspectos: Mobilidade, veracidade, benevolência, racionalidade e cooperação. As definições são:

- Mobilidade: O Agente deve pode mover-se no ambiente, por exemplo, em uma rede.
- Veracidade: Agente não comunica informações falsas.
- Benevolência: Agente ajudará os outros.
- Racionalidade: O agente não irá agir de forma a impedir a realização de seus objetivos.
- Cooperação: O agente coopera com o usuário.

2.3.3 Arquitetura de agentes

A arquitetura de agentes varia de acordo com a complexidade da sua autonomia, ou seja, com a capacidade de reagir aos estímulos do ambiente. Conforme verificado em [17], os tipos de arquitetura são: orientadas à tabela, reflexiva simples, reflexiva baseado em modelo, baseada em objetivo, baseada em utilidade.

A primeira arquitetura a ser explorada é o agente orientado à tabelas. Todas as ações dos agentes dessa arquitetura são conhecidas e estão gravadas em uma tabela. Assim, quando o agente receber o estímulo ele já terá a ação a ser tomada previamente gravada em sua memória. Logo para construir esse tipo de agente, fica claro que além de saber todas percepções possíveis, será necessário definir ações apropriadas para todas. Isso levará a tabelas muito complexas e o tamanho pode facilmente passar a ordem de milhões dependendo do número de entradas.

A arquitetura reflexiva simples é um dos tipos mais simples de agente. Nele, o agente seleciona a ação com base unicamente na percepção atual, desconsiderando assim uma grande tabela de decisões. A decisão é tomada com base de regras condição-ação: Se uma condição ocorrer, uma ação será tomada. Por exemplo, vamos supor um agente médico que determina o diagnóstico de uma doença no paciente caso exista alguma anomalia no organismo (Por exemplo, paciente com febre). Uma condição-ação poderia ser:

if anomalia-organismo then diagnóstico-médico

Esse tipo de agente é bastante simples, o que é uma vantagem comparado à arquitetura de tabela. Porém, essa abordagem requer um ambiente totalmente observável, visto que esse tipo de agente possui uma inteligência bastante limitada. No exemplo do agente médico existem diversas maneiras de se detectar uma anomalia no organismo do paciente, seria necessário conhecer todas as formas para usarmos uma abordagem reativa simples.

A arquitetura reflexiva baseada em modelos funciona de maneira similar a anterior. Nessa abordagem, é levado em conta a parte do ambiente que não é visível neste momento. E para saber o "momento atual" de um agente, é necessário guardar a informação de estado consigo. Para atualizar o estado do agente, é necessário conhecer como o mundo desenvolve-se independente do agente (no caso do exemplo, como o organismo funciona) e é necessário saber as ações dos agentes no ambiente. Esses dois conhecimentos do ambiente

são chamados de **modelo do mundo**. O agente que usa esse tipo de abordagem é chamado de agente baseado em modelo.

Na arquitetura reflexiva baseada em objetivo, as ações do agente são tomadas apenas se o aproximam de alcançar um objetivo. Para isso, será necessário algo além do estado atual do ambiente: Será necessário informações do objetivo a ser atingido. Assim o agente pode combinar as informações do estado e o objetivo para determinar se deve ou não agir sobre o ambiente. Essa arquitetura porém é obviamente mais complexa e de certa forma ineficiente. Porém ela permite uma maior flexibilização das ações em determinados ambientes, visto que suas decisões são representadas de forma explícita e podem ser modificadas. É interessante notar que esse tipo de arquitetura não trata ações com objetivos conflitantes.

E por fim, a arquitetura reflexiva baseada em utilidade não utiliza apenas objetivos para realizar a próxima decisão, mas dá ao agente a capacidade de fazer comparações sobre o estado do ambiente e as ações a serem tomadas: Quais delas são mais baratas, confiáveis, resilientes e rápidas do que as outras. A capacidade de avaliação do agente é chamada de função de utilidade, que mapeia uma sequência de estados em um número real que determina o grau de utilidade. Esse mecanismo possibilita a decisão racional de escolha entre vários objetivos conflitantes. Por exemplo, escolher entre um objetivo mais barato ao invés de escolher entre o mais rápido.

2.3.4 Sistemas Multiagentes

Sistemas multiagentes são sistemas compostos por vários agentes capazes de se comunicar, possuindo uma linguagem de alto nível para isso. Um agente possui um objeto que, normalmente, é distinto dos objetivos de outros agentes e pode ou não cooperar com outros agentes para a realização de uma tarefa.

De acordo com [20], podemos encontrar as seguintes características principais de ambientes em SMAs:

- Ambientes SMAs fornecem protocolos específicos para comunicação e interação. Cada ambiente tem as suas particularidades: Alguns são em uma única máquina, outros são compartilhados com o mundo real e outros são distribuídos. Cabe a cada ambiente definir um protocolo onde todos agentes devem obedecer para comunicar-se.
- SMAs são tipicamente abertos.
- SMAs contém agentes que são autônomos e individualistas.

É trivial imaginar que um sistema multiagente é designado para a solução de problemas de forma distribuída, onde o problema é distribuído entre os agentes que, juntos, trabalham cooperativamente e concorrentemente para a resolução deste problema.

O termo *cooperação* é normalmente associado ao mundo de sistemas concorrentes. É importate notar a distinção desses conceitos na literatura de sistemas multiagentes. De acordo com [22], existem duas principais diferênças dos conceitos entre as duas situações.

A primeira delas é que agentes são designados de forma diferente, com objetivos diferentes. Em um ambiente com vários agentes, eles devem trabalhar estratégicamente

para alcançar seus objetos. Conforme dito anteriormente, a possibilidade de agentes não cooperarem é perfeitamente plausível.

A segunda diferença está no ponto onde um agente age de forma autônoma, ou seja, ele toma suas próprias decisões sem interferências de outros agentes, tomando-as em tempo de execução. Logo um ecossistema de agentes deve ser capaz de coordenar dinâmicamente (em tempo de execução) suas ações, cooperando com outros agentes para atingir os objetivos. Em aplicações distribuídas, esses comportamentos já são desenhados durante o planejamento do software.

A forma de agentes resolverem problemas foi baseada na técnica distribuição cooperativa de resolução de problemas - *cooperative distributed problem solving* (CDPS). De acordo com [11], a técnica CDPS consiste de uma rede de baixo acoplamento provida de sofisticados nós resolvedores de problemas que precisam cooperar entre si, pois nenhum deles possui recursos, informações e *expertise* suficientes para resolver algum problema sozinho. Cada nó possui uma *expertise* diferente que pode resolver parte do problema.

Inicialmente essa técnica assumiu que os problemas fossem de ordem benevolente. Isso significa que, implicitamente, os agentes compartilham o mesmo objetivo de resolver o problema proposto. Isso implica que todos os agentes ajudarão sempre que possível, mesmo tendo prejuízos na execução da ação, pois o objetivo geral de todos será a resolução do problema maior. Esse cenário é plausível desde que uma organização ou entidade tenha o controle de (ou modele) todos os agentes.

Em um cenário mais realista (e de maior enfoque dos estudos de SMA), agentes podem pertencer à sociedades com interesses próprios, diferentes de outras sociedades. Logo, é possível ocorrer o conflito de interesses neste ambiente, situação que força os agentes a cooperarem com os outros para alcançarem seus objetivos.

O processo CDPS pode ser divido em três etapas:

- A decomposição do problema.
- Solução do subproblema
- Integração da solução

Na primeira etapa, o problema é quebrado em instâncias menores e dividido entre os agentes. Esses subproblemas podem ser quebrados em diversas subpartes, permitindo uma maior decomposição e consequentemente um maior número de agentes trabalhando na resolução.

Em seguida, os subproblemas são resolvidos pelos agentes. Isso pode significar troca de informações entre os agentes, com os que resolveram as suas instâncias compartilhando informações com outros agentes.

Por fim as soluções dos subproblemas são integradas em uma resolução completa do problema original.

Com uma solução compartilhada de resolução de problemas, a arquitetura de sistemas multiagentes mostra-se bastante robusta neste quesito. É necessário porém saber dos detalhes da comunicação entre os agentes, a forma de envio das mensagens, as suas linguagens, bem como outras particularidades.

2.3.5 Comunicação

A comunicação é um dos aspectos mais importantes no desenvolvimento de SMAs. Problemas de sincronização entre as partes que se comunicam devem ser devidamente estudados. A situação mais simples possível da comunicação, onde o agente A envia uma mensagem ao agente B que está prontamente disponível para receber a mensagem nem sempre é a o cenário mais recorrente. Para tanto, é necessário entender os pormenores da comunicação em um Sistema Multiagente.

Em uma aplicação normal (Desktop ou Web), a comunicação entre objetos pode ser mais simplificada. Por exemplo, supondo uma aplicação em que existe dois objetos, a e b e que o objeto a tenha um método público chamado *m1*. O objeto b pode ser comunicar com o objeto a por meio do método *m1*, provavelmente da seguinte forma *a.m1(args)*, onde *args* são os argumentos enviados ao objeto a e a sintaxe pode ser diferente da apresentada, dependendo da linguagem de programação. É importante notar que o controle da execução do método *m1* não está no objeto a, mas sim no objeto b, que decide o momento o qual o método será invocado.

Esse cenário de comunicação é diferente em um ambiente SMA. Supondo dois agentes a e b, onde o agente a tem a capacidade de executar a ação α. O agente b não poderá invocar diretamente o método que corresponde à ação α, visto que os agentes são autônomos e independentes: Cada um tem somente tem total controle sobre suas ações e seus estados. O agente precisará enviar a solicitação da execução da ação α por meio de mensagem. Isso porém não garante que o agente a executará esta ação, pois pode não ser do seu interesse. Os agentes podem também influênciar o comportamento de outros agentes, alterando seu estado interno para a execução de ações e cooperando para o cumprimento do objetivo de outros agentes.

A comunicação dos agentes é baseada na teoria dos atos de fala (do inglês *Speech act theory*) e trata a comunicação como uma ação. A teoria dos atos de fala, publicada em 1962 [6] por John Austin, onde ele percebe que certas expressões de linguagem natural, ou atos de fala, possuem a característica de realizar ação em um interlocutor, causando assim uma mudança de de estado da mesma forma que uma ação física. Logo, as expressões descrevem as ações por meio de desejos, habilidades e crenças.

De acordo com [21], a teoria dos atos de fala possuem duas características:

- A distinção entre um o significado expressado por uma expressão e a forma como essa expressão é utilizada.
- Expressões de todos os tipos podem ser considerados como atos, pois mudam o mundo de alguma forma.

Posteriormente o trabalho de John Searle [19], relacionado ao de Austin, separa uma ação de um ato entre orador(*speaker*) e ouvinte(*hearer*) identifinca propriedades e conduções que um discurso deve conter para realizar ações sucedidas. Além disso, ele classifica alguns atos de discursos em 5 classes:

- Representativas - Representa o ato de um orador representar uma verdade para o ouvinte. Pode ser entendido como uma ação de informar(*inform*).
- Diretivas - Tentativa do orador de fazer algum ouvinte realizar alguma ação por meio do seu ato.. Pode ser entendido como uma ação de requisição(*request*).

- Comissivas - O orador toma alguma atitude em relação à uma ação em andamento.
- Expressivas - Expressa algum estado psicológico.
- Declarações - Causa algum efeito relacionado a determinado assunto.

A comunicação então baseia-se nos atos de fala para prever a interação com seres humanos e é definida por meio de semânticas definidas pela teoria da Inteligência Artificial [22]. O formalismo para a comunicação foi escolhido de forma que foi possível representar os atos de discursos em uma lógica multimodal, que contém os operadores de desejos, habilidades e crenças dos atos de discurso.

Da mesma forma que a teoria dos atos de fala influênciou na arquitetura da comunicação, ela influênciou também nas várias linguagens de comunicação dos agentes. Linguagens foram desenvolvidas para, não apenas representar ações de agentes, mas também para representar o conhecimento entre sistemas autônomos. No início dos anos 90, duas linguagens foram desenvolvidas pelo consórcio *Knowledge Sharing Effort*, encabeçados pela agência norte americana *Defense Advanced Research Projects Agency*(DARPA) [2].

- *Knowledge Query and Manipulation Language* (KQML) - Protocolo designado para a comunicação de agentes, em uma arquitetura que esses agentes sejam projetados para resolver problemas da arquitetura cliente-servidor [16]. Não existe o foco com o conteúdo da mensagem.
- *Knowledge Interchange Format* (KIF) - Criada para facilitar a troca de conhecimento entre agentes, suas declarações são providas de significados que podem ser compreensíveis a qualquer agente que conheça a estrutura da linguagem. Não possui foco na transmissão da mensagem [13].

KQML

A linguagem KQML define um protocolo para comunicação de agentes, onde cada mensagem tem um enunciado performativo (*performative*), que varia com o seu objetivo, e em seguida os parâmetros da mensagem. O KQML define vários enunciados performativos que distinguem-se pelo objetivo da mensagem, sendo dividas em três categorias: Discursivas, intervenção/mecânica e facilitação e *networking*. Por exemplo, uma mensagem com o tipo performativo *ask-one* indica que o agente remetente A deseja saber uma resposta do agente B sobre o conteúdo da mensagem. Em [16], é possível verificar que a última versão da linguagem define mais de trinta enunciados performativos.

Os maioria dos parâmetros são opcionais, sendo os mais importantes: *content* e *receiver*. A tabela 2.3 lista os principais parâmetros em uma mensagem nesta linguagem, bem como seu significado.

O formato da mensagem é completamente compreensível aos humanos. Na mensagem 2.1 podemos verificar na primeira linha o enunciado performativo *ask-one*, onde será uma mensagem de consulta. Nas linhas abaixo, visualizamos todos os parâmetros da mensagem antecedidos por (:). O primeiro parâmetro da mensagem é *receiver* como controle-estoque, ou seja, esse será o destinatário da mensagem. O segundo parâmetro *language* tem o valor PROLOG indicando que a sintaxe do conteúdo está escrita em PROLOG. O próximo parâmetro, *ontology* informa a ontologia que espressa o conteúdo.

Tabela 2.3: Listagem de atributos de uma mensagem em KQML

Parâmetro	**Significado**
sender	Remetente da mensagem.
receiver	Destinatário da mensagem.
reply-with	Identifica se o remetente espera uma respota. Em caso positivo, o campo *in-reply-to* deve ser preenchido com a referência para a resposta.
in-reply-to	Campo conténdo a referência para a resposta solicitada.
language	Linguagem em que o campo *content* está escrito.
ontology	Indica a forma que deve ser interpretada o conteúdo do campo *content*.
content	Conteúdo da mensagem.

Por fim, o parâmetro *content* que indica o conteúdo da mensagem, no caso, uma consulta escrita em PROLOG perguntando pelo preço de um computador.

Listing 2.1: Exemplo de mensagem em KQML

```
(ask-one
        :receiver controle-estoque
        :language PROLOG
        :ontology PRODUTOS
        :content  (PRECO COMPUTADOR ?price)
)
```

Um exemplo de diálogo escrito em KQML pode ser visto na sequência de mensagens 2.2

Listing 2.2: Exemplo de diálogo em KQML

```
(evaluate
        :sender A
        :receiver B
        :language PROLOG
        :ontology PRODUTOS
        :reply-with q1
        :content  (PRECO COMPUTADOR ?price)
)
(reply
        :sender B
        :receiver A
        :language PROLOG
        :ontology PRODUTOS
        :in-reply-to q1
        :content  (=2000.00)
)
```

A primeira mensagem do diálogo possui o enunciado performativo é *evaluate*, significando que o emissor A deseja avaliar o conteúdo com B. Nos parâmetros é possível notar que a linguagem da mensagem é PROLOG, utiliza a ontologia PRODUTOS e o conteúdo

é uma consulta em prolog. O parâmetro *reply-with* cria uma referência para a consulta do conteúdo conteúdo.

Na segunda mensagem, o seu enunciado performativo é *reply*, significando uma mensagem do tipo resposta. O parâmetro *in-reply-to q1* especifica essa mensagem como resposta à q1, ou seja, à consulta da mensagem anterior. Dessa forma a linguagem consegue distinguir respostas de um mesmo remetente. Os outros parâmetros são o emissor B, destinatário A, linguagem PROLOG e o conteúdo da mensagem, o valor da consulta q1.

De acordo com [22], a adoção desta linguagem pela comunidade de SMA foi significante, mas sofreu diversas críticas:

- A fluídez e a não restrição do KQML fez com que diversas implementações extendidas surgissem, impossibilitando a interoperabilidade entre sistemas.
- Mecanismos de transporte do KQML nunca foram bem definidos, causando problemas de diversas implementações destes mecanismos e prejudicando novamente a interoperabilidade.
- A semântica do KQML nunca foi formalmente definida, ocasionando em má interpretações dos enunciados performativos.
- A linguagem não possui enunciados performativos adequados para algumas semânticas. Por exemplo, a inexistência do enunciado *comissives*.

Dessa forma, novos desenvolvimentos de linguagens fizeram-se necessários.

KIF

A linguagem foi desenvolvida para expressar conhecimento a cerca de um determinado domínio, sendo possível assim a troca de conhecimentos entre agentes. De acordo com [13], a linguagem possui as seguintes características:

- Tem uma semântica declarativa, sendo possível entender o seu significado sem a necessidade de um interpretador para manipulação das expressões.
- É logicamente compreensível.
- É provida com a capacidade de reproduzir meta-conhecimento, ou seja, conhecimento a respeito da representação do conhecimento. Com isso, é possível reproduzir novas representações de conhecimento sem a necessidade de modificar a linguagem.

A linguagem é baseada na lógica de primeira ordem onde são definidos operadores como existe($\exists$) e para todo($\forall$),possibilitando aos agentes a expressão de diversas propriedades, domínios, dentre outros. Além disso, ela define um vocabulário básico para a expressão de tipos básicos (números, caracteres, strings) e algumas funções padrões para lidar com esses tipos de dados (menor que, maior que, soma, dentre outros).

O trecho de código 2.3 ilustra um exemplo de expressão na linguagem KIF, validando que a temperatura de m1 é 83 graus Célsius.

Listing 2.3: Exemplo de expressão de conteúdo com a linguagem KIF. Fonte: [22]

```
(= (temperature ml ) ( scalar 83 Celsius))
```

2.3.6 Linguagem de Comunicação de Agentes FIPA

Após as críticas à linguagem KQML, o consórcio *Foundation for Intelligent Physical Agents* (FIPA) começou a trabalhar em 1995 no desenvolvimento da padronização de SMAs. O núcleo dessa padronização foi o desenvolvimento de uma linguagem de comunicação de agentes (ACL) padronizada para todas as plataformas.

Baseado na linguagem KQML, a estrutura das mensagens é a mesma e os seus atributos são semelhantes. A maior diferença entre as duas linguagens são os enunciados performativos. Foram definidos 20 tipos de mensagens performativas, muito semelhante ao KQML, porém definindo formalmente as interpretações dessas mensagens e não definindo nenhuma linguagem para o conteúdo da mensagem.

Devido ao fato da linguagem KQML não ter performativos adequados, a ACL da FIPA recebeu total preocupação na definição formal da semântica da linguagem. A tabela 2.4 contém um breve resumo dos enunciados performativos disponíveis em [1]. É importante notar que a especificação [1] possui toda a descrição formal de cada enunciado performativo aqui descrito.

As performativas *request* e *inform* consideradas principais pela especificação da FIPA, pois orientam toda a linguagem de comunicação. Além disso, é possível derivar as outras performativas por meio delas.

2.3.7 Ontologias

Ontologia é um ramo da Filosofia que dedica-se a estudar e representar a natureza do ser, existência ou realidade. É um conceito formalização dos conceitos e relacionamentos que podem existir em um determinado universo. Para a Inteligência Artificial, é tudo aquilo que pode ser representado. De acordo com o autor [17], a representação de conceitos e objetos pode ser entendida como Engenharia Ontológica:

> *...concentrating on general concepts-such as Actions, Time, Physical Objects, and Beliefs - that occur in many different domains. Representing these abstract concepts is sometimes called ontological engineering.*

A possibilidade de representação de conhecimento por meio de ontologias é bastante vasta. A sua forma de especificação pode ser entendida como uma hierarquia, onde os conhecimentos são organizados na forma de árvore, possibilitando a inserção de novos conhecimento a qualquer momento.

A imagem 2.5 mostra a organização geral de conceitos, chamado de ontologias superiores. As ontologias gerais estão representadas no topo da árvore, e as suas especialidades vão crescendo no sentido das folhas.

Em um ambiente de Sistemas Multiagentes, além de especificar uma linguagem para comunicação, dois agentes podem comunicar-se com relação à um determinado domínio de aplicação: Podem negociar valores de carteiras em uma organização financeira, podem trocar mensagens sobre os dados analisados de performance de veículos, dentre outros exemplos. Em outras palavras, dois agentes podem comunicar-se usando a mesma ontologia.

Existem muitas linguagens que foram desenvolvidas para expressar ontologias. A mais comum delas é a *eXtensible Markup Language* (XML), uma linguagem de marcação que organiza os dados de forma hierarquizada.

Tabela 2.4: Listagem de Enunciados Performativos

Tipo	**Descrição**
Accept Proposal	Declaração de aceite de proposta feita por um agente.
Agree	Performativa feita por um agente indicando que aceitou o *request* feito por outro agente.
Cancel	Cancela uma mensagem de *request*, informando que não irá mais participar daquela conversação.
Call for Proposal	Performativo que indica o início de uma negociação entre agentes.
Confirm	A confirmação permite ao emissor da mensagem confirmar a veracidade do conteúdo da mensagem.
Disconfirm	Similar à confirmação, porém informando ao emissor da não certeza do conteúdo da mensagem.
Failure	Permite ao agente indicar que a tentativa de executar uma ação falhou.
Inform	Informa à um destinatário que o conteúdo da mensagem é verdadeiro, implicando que o remetente da mensagem também acredita no seu conteúdo. É uma das mais importantes performativas feitas pela FIPA.
Inform If	Envia uma mensagem a qual o seu conteúdo pode ser verdadeiro ou falso.
Inform Ref	Similar ao *Inform If*, com a diferença que ao invés de questionar se é verdadeiro ou falso, ele solicita o valor de uma expressão para o remetente.
Not Understood	Informa à um agente que não entendeu por que determinada ação deve ser realizada. Usada quando o estado interno do agente não é compatível com a mensagem.
Propagate	Consiste em propagar o conteúdo da mensagem para um grupo de agentes.
Propose	Permite um agente realizar uma proposta em uma negociação para outro agente.
Proxy	Permite ao destinatário da mensagem agir como um *proxy* para os agentes que estão descritos no conteúdo da mensagem.
Query If	Permite à um agente consultar um destinatário sobre a validade do conteúdo.
Query Ref	Similar ao *query-if*, porém o agente remetente irá consultar o valor de uma expressão
Refuse	Indica que um determinado agente não irá executar uma ação que foi determinada por outro agente.
Reject Proposal	Permite um agente rejeitar uma proposta feita por outro agente em uma negociação.
Request	Permite à um agente requisitar que outro agente execute determinada ação.
Request When	Permite à um agente requisitar que outro agente execute determinada ação quando a proposição (no conteúdo da mensagem) for verdadeira.
Request Whenever	Similar ao *request-when*, porém o agente nunca irá executar a ação quando a proposição (no conteúdo da mensagem) for verdadeira.
Subscribe	O conteúdo será uma proposição e o remetente da mensagem será notificado sempre que essa proposição for verdadeira.

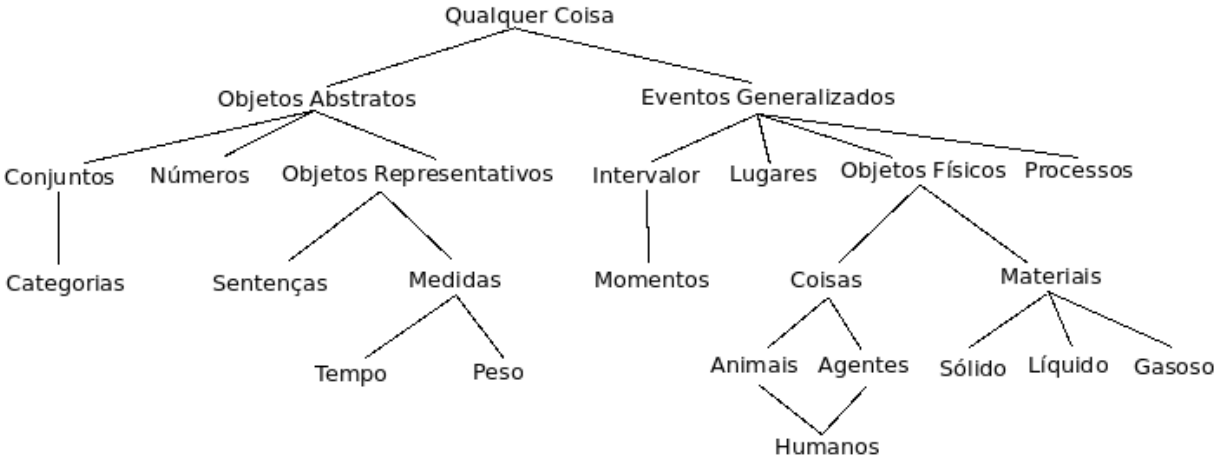

Figura 2.5: Ontologias superiores do mundo, cada uma indicando um conceito ou especialização do seu superior.

O código 2.4 possui um exemplo de ontologia escrito na linguagem XML que representa o domínio de países. Nele podemos ver a ontologia superior, País. Cada país é composto por um conjunto de estados, conténdo as propriedades nome e sigla. Cada estado possui um conjunto de cidades. Cada cidade possui os atributos nome e população. Esses atributos poderiam ser escritos de forma diferente no trecho 2.4, porém sem alteração de semântica.

Listing 2.4: Exemplo de código XML representando uma ontologia simples de cidades.

```
<Pais nome="Brasil">
        <Estado nome="Goias" sigla="GO">
                <Cidade>
                        <Nome>Goiania</Nome>
                        <Populacao>1.300.000</Populacao>
                </Cidade>
                <Cidade>
                        <Nome>Anapolis</Nome>
                        <Populacao>342.347</Populacao>
                </Cidade>
                <Cidade>
                        <Nome>Aparecida de Goiania</Nome>
                        <Populacao>474.219</Populacao>
                </Cidade>
        </Estado>
        <Estado nome="Sao Paulo" sigla="SP" >
                <Cidade>
                        ...
                </Cidade>
                ...
        </Estado>
        ...
```

<Pais>

2.4 Multiagent Systems Engineering - MASE

Atualmente muitas abordagens de projeto de desenvolvimento de software conseguem com sucesso definir uma metodologia para construção, implantação e manutenção do software. A definição dessas engenharias de software possuem diversos casos de sucesso em sua maioria nas áreas de Análise de Projetos Orientados à Objetos. O advento de sistemas multiagentes trouxe à tona a necessidade de outras metodologias de desenvolvimento diferente daquelas, visto a diferença de abordagem no software.

Pela característica autônoma dos agentes, não é possível compará-los à simples objetos que serão invocados por outros objetos. Não existe interação direta, apenas coordenação de ações via conversação para cada agente atingir suas próprias métas.

O desenvolvimento de uma metodologia para projetar sistemas multiagentes surgiu, chamada *Multiagent Systems Engineering* (MASE), onde os requisitos e metas do SMA são levantados e a partir de então as tarefas e agentes são projetadas para lhes atender. O MASE utiliza-se de alguns modelos gráficos para a descrição dos agentes, seus objetivos, suas interações e a sua arquitetura.

De acordo com [18], a metodologia do MASE é baseada nas mais tradicionais metodologias de desenvolvimento de software, dividida em duas fases principais: Análise e Design.

A primeira fase, chamada de análise, consiste no levantamento e entendimento dos requisitos com o objetivo de um conjunto de regras, as quais são associadas à tarefas que devem ser realizadas para o sistema atingir seus objetivos. No fim dessa fase, alguns artefatos são gerados que nortearão a próxima etapa da metodologia. A fase de análise pode ser dividida nos seguintes três passos:

- Capturar Metas
- Desenvolver Casos de Uso
- Refinar Regras

A fase de design consiste na modelagem do SMA de acordo com as regras levantadas na fase anterior. O objetivo é a definição das conversações que existirão entre os agentes, bem como a arquitetura geral do sistema. Essa fase pode ser divida em quatro passos:

- Criar as Classes dos Agentes
- Construir Conversações
- Montagem dos Agentes
- Design do Sistema

2.4.1 Análise

De acordo com [18], a fase de análise objetiva o desenvolvimento de um conjunto de regras, que descrevem uma funcionalidade para uma entidade, as quais as atividades descrevem o que deve ser feito para o sistema atingir o seus objetivos.

Captura de Metas

O primeiro passo dessa fase, Captura de Metas, consiste na transformação da especificação inicial do sistema em um conjunto estruturado de metas. Entende-se que as *metas* que o sistema leva em consideração são relacionadas ao sistema (e não ao usuário), visto que são mais estáveis e as metas do sistema devem satisfazer, de forma geral, os objetivos do usuário [18].

Dividida em dois subpassos, o primeiro deles é a identificação de todas as metas. A partir da documentação inicial do sistema, conténdo os requisito funcionais, são extraídos os objetivos de cada cenários do sistema. É importante ressaltar que as metas devem descrer de forma geral e sucinta o comportamento do sistema, descrevendo o que ele deve fazer e não como deve ser feito.

O segundo subpasso consiste na estruturação de metas em forma hierarquizada. É necessário separar quais metas são mais abstratas e de que forma elas podem ser agrupadas em forma de hierarquia. Com isso, eliminam-se algumas metas que são repetidas e identificam-se quais metas são atingidas por meio de outras submetas.

Desenvolver Casos de Uso

Neste passo o objetivo é entender o comportamento e o fluxo de execução do sistema por meio dos casos de uso, além de haver um entendimento maior sobre como o sistema irá se comunicar. Para tanto, este passo visa o levantamento e criação dos casos de uso do sistema, bem como o diagrama de sequência para detalhar a ordem dos eventos de cada cenário.

Os casos de uso geralmente são levantados a partir dos requisitos iniciais do sistema. Nele, são identificados os participantes (atores) e a sua interação com o sistema, esclarecendo a comunicação de alguns módulos do sistema.

O diagrama de sequência define os eventos que cada interação pode criar, mostrando a ordem de execução destes eventos e a sua comunicação. Esses diagramas são criados para cada caso de uso, podendo haver mais de um para cada caso de uso. O objetivo principal é o levantamento dos eventos e das regras.

Refinar Regras

O último passo da fase de análise, o refinamento de regas consiste na associação metas e seus diagramas de sequências às regras e suas respectivas tarefas. De acordo com [18], a associação de tarefas à regras é a melhor forma de modelagem de Sistemas Multiagentes.

Em geral, a associação de metas às regras é de um para um, não sendo uma regra necessáriamente. Durante esta etapa, algumas considerações relativas ao desenho do sistema devem ser levadas em consideração. Caso exista alguma alteração (adição de uma nova meta, alteração de caso de uso, dentre outras), a metodologia permite que o analista retorne aos passos anteriores e remodele a solução. Algumas metas podem ser combinadas em apenas uma regra, simplificando o desenho do sistema.

A interface com sistemas externos geralmente é tratada como uma regra diferente, visto que sua complexidade pode variar. O MASE não modela explicitamente um ator humano, pois o considera como um ator externo ao sistema.

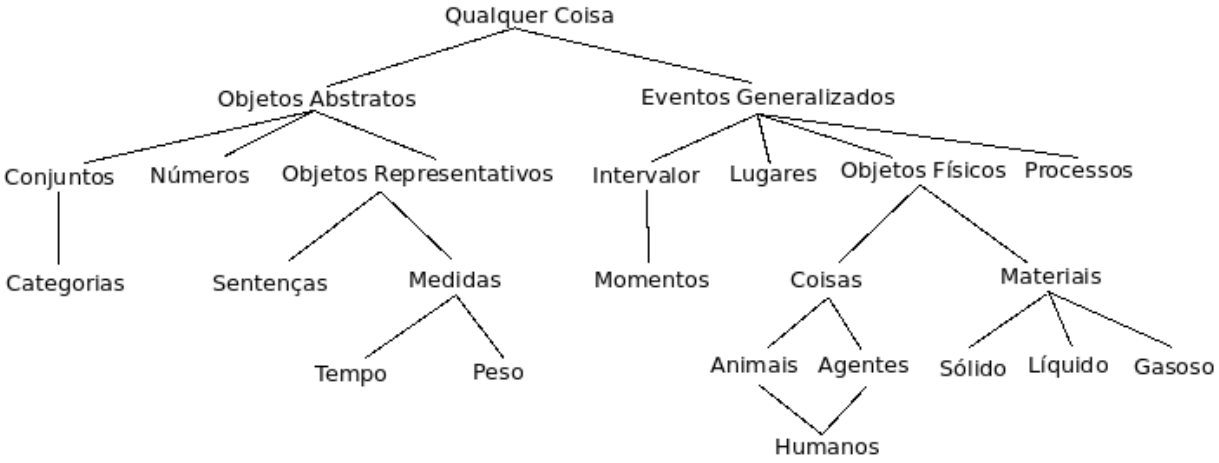

Figura 2.6: Representação utilizada no MASE Role Model.

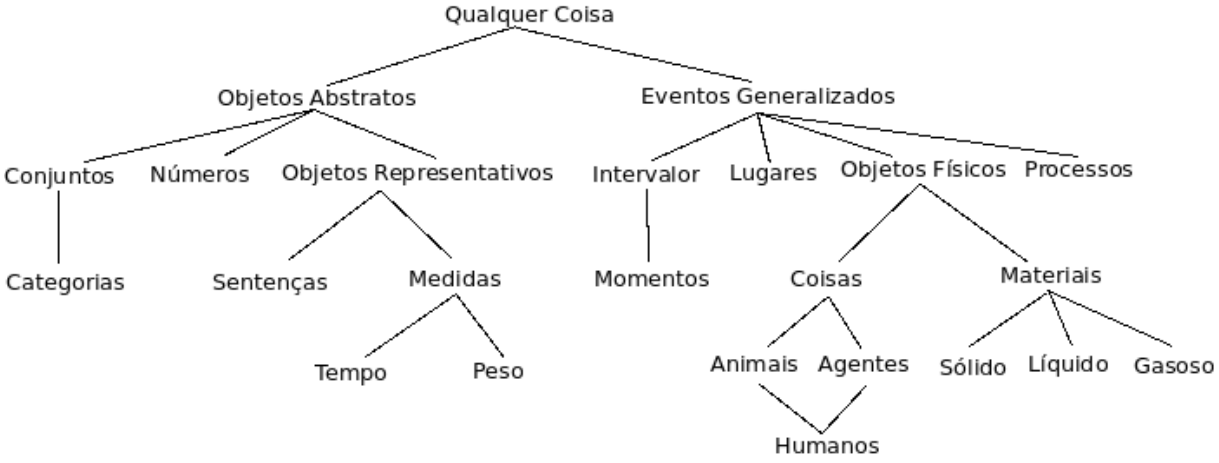

Figura 2.7: Representação utilizada no *Concurrent Task Diagram*.

Após a associação das regras, elas são estruturas em uma modelagem chamada MASE Role Model, conténdo informações de interações entre as tarefas. A imagem 2.6 representa a notação do Mase Role Model utilizada por [18]. Os retângulos são as regras, as elipses são as tarefas e as setas entre as tarefas representam as suas comunicações. Essas comunicações podem ser por meio de mensagens, caso as regras estejam separadas em agentes diferentes.

Caso as regras compartilhem tarefas será necessário remodelar a composição da regra, visto que o MASE não permite a duplicação de tarefas.

De forma geral, cada regra possui uma série de tarefas que podem (ou não) executar paralelamente. Cada tarefa possui um comportamento que pode depender de outras regras para o cumprimento de seu objetivo. Preocupando-se com as conversações entre tarefas, o MASE determina neste passo a criação do *Concurrent Task Diagram*.

O diagrama é representado por meio de autômatos de estados finitos, devido a facilidade de construção e entendimento. A imagem 2.7 é uma representação deste diagrama, representando transições de estados. A transição consiste de uma mudança de estado do agente, podendo envolver um processamento ou uma comunicação externa.

A mudança de estado do automato é equivalente à mudança de estado do agente. O trecho 2.5 representa a sintaxe da transição de estado no diagrama.

Listing 2.5: Sintaxe da mudança de estado.

```
trigger [guard] ^ transmission(s)
```

O *token* trigger representa um evento que inicia a mudança de estado, geralmente vindo de outra tarefa da mesma regra. O *token [guard]* é a verificação da validade do código *guard*, ocorrendo a mudança de estado somente quando a condição for verdadeira. Por fim ocorrem as transmissões, que podem conter o parâmetro *s*.

Para comunicações externas, dois eventos especiais foram definidos: O evento *send* indicando o envio de mensagem e o evento *receive*, indicando o recebimento de mensagem.

A mensagem sempre possui um cabeçalho performativo (definida pela FIPA, representa o objetivo da mensagem) com a seguinte sintaxe:*performative(p1...pn)*, onde *p1...pn* indicam os parâmetros da mensagem.

Após a definição dos *Concurrent Tasks Diagrams*, o analista pode combinar tarefas, a fim de diminuir a complexidade do sistema.

Com isso, o sistema já possui a complexidade das regras determinadas, bem como as tarefas necessárias para atingir seus objetivos.

2.4.2 Design

A fase de design é divida em quatro passos: Criar Classes dos Agentes, Construir Conversações, Montagem dos Agentes e Design do Sistema. O principal objetivo desta fase é projetar os agentes e suas interações de acordo com os insumos construídos na fase anterior.

Criando as Classes dos Agentes

Neste passo, os agentes são criados com base nas regras definidas na fase anterior. Para cada agente criado deve existir pelo menos uma regra associada, caso contrário o levantamento de regras mostra-se incompleto. Dessa forma, o MASE garante que todas os objetivos do sistema são atingidos, já que as regras do sistema estão relacionados com as metas que foram levantadas na etapa anterior.

Ao fim deste passo é necessário a criação de um novo diagrama, o *Agent Class Diagram*. Nele, as classes dos agentes são associadas com as regras levantadas e as comunicações entre as classes. A imagem 2.8 possui a notação do diagrama utilizada por [18].

Construir Conversações

A próxima etapa da segunda fase diz respeito às conversações que existirão no SMA. Aqui, cada detalhe da conversação entre dois agentes deverá ser planejado para que haja o cumprimento da meta do sistema.

De acordo com [18], quando um agente recebe uma mensagem, ele compara com as suas conversações ativas. Caso ele encontre alguma, o agente muda o seu estado e realiza as ações relativas para atingir esse estado. Caso contrário, é assumido que o agente emissor deseja iniciar uma nova conversação e o receptor compara com as suas possibidades de tipos de conversação disponíveis para participar.

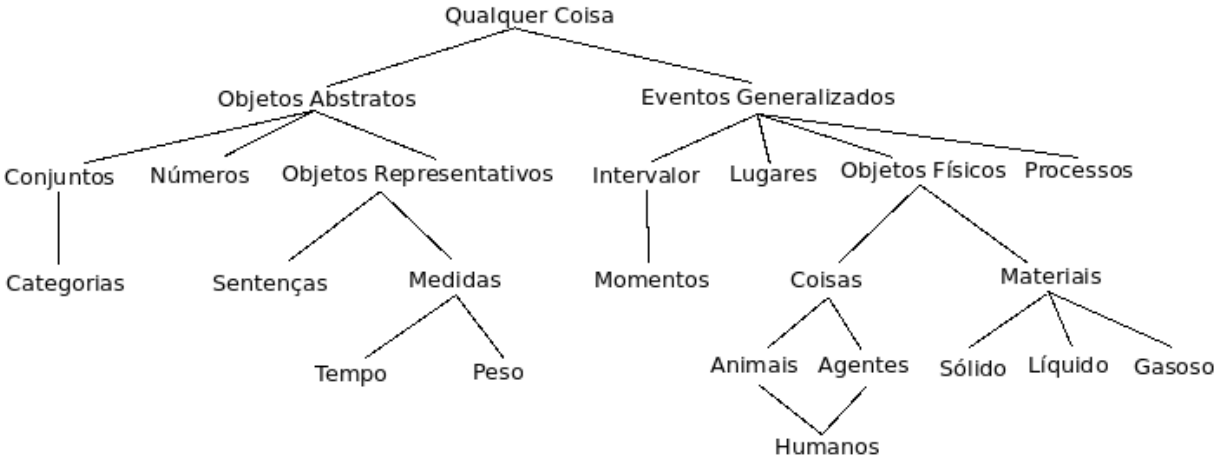

Figura 2.8: Representação utilizada no *Agent Class Diagram*.

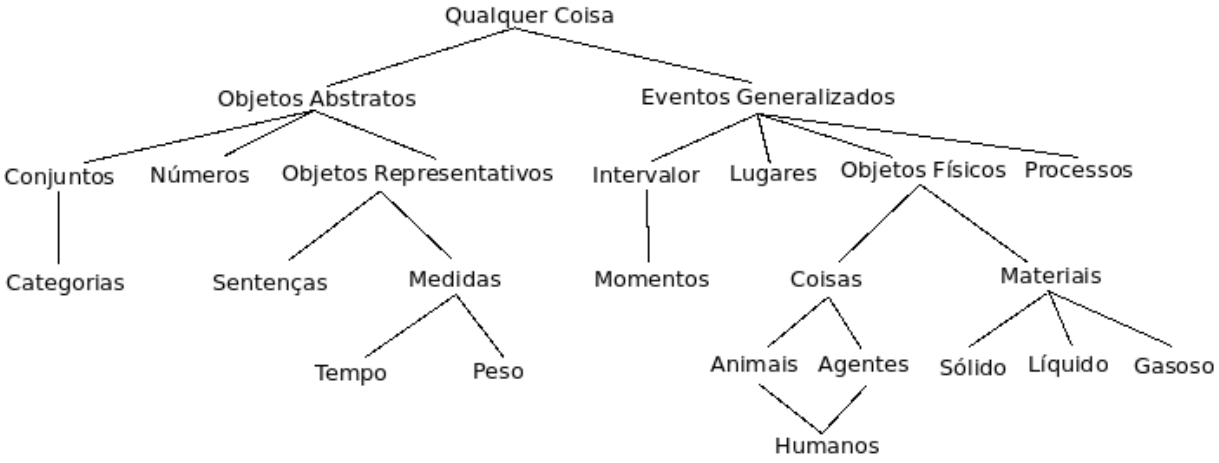

Figura 2.9: Representação utilizada no Diagrama de Comunicação.

Neste passo, é criado o Diagrama de Comunicação. Nele, as conversações são montadas por meio do mesmo autômato de estados finitos, utilizado na fase análise, passo *refinar regras*. Os estados do aumtômato são as ações que devem ser realizadas pelo agente. As transições de estado são as conversações que são feitas pelo agente.

O trecho 2.6 mostra a sintaxe usada na conversação de agentes. O *token rec-mess* indica que a mensagem com os parâmetros *args1* foi recebida caso a condição *cond* seja verdadeira. Então o método *action* é chamado e a mensagem *trans-mess* com os argumentos *args2*. Todos os elementos da mensagem são opcionais.

Listing 2.6: Sintaxe da conversação entre dois agentes.

```
rec-mess(args1) [cond] / action ^ trans-mess(args2)
```

A imagem 2.9 representa um exemplo da sintaxe do Diagrama de Comunicação.

Montagem dos Agentes

Neste passo, o estado interno dos agentes é criado. É necessário nesta etapa definir a arquitetura dos agentes e os componentes que irão compor esta arquitetura. Para

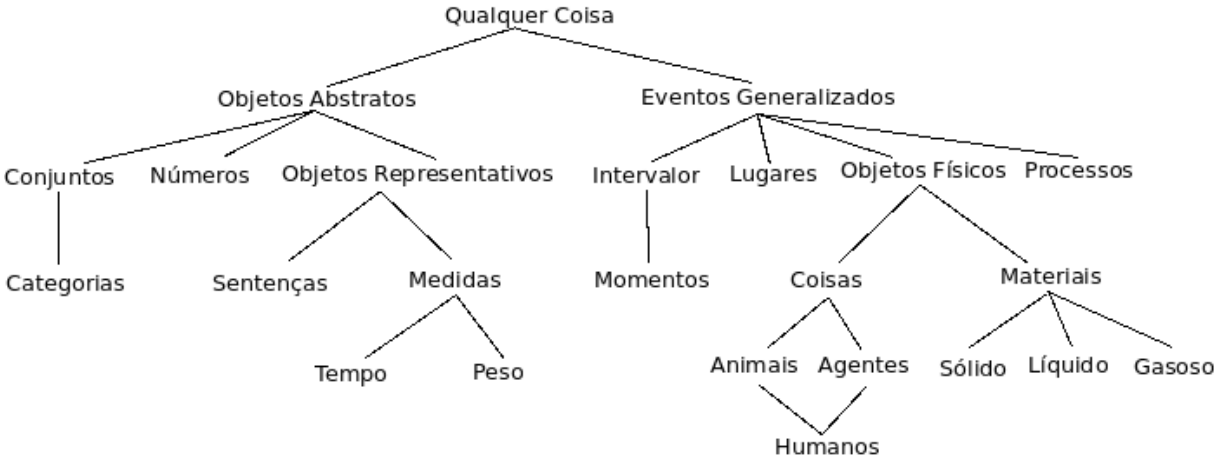

Figura 2.10: Notação utilizada na arquitetura de agentes.

isso, é necessário criar o Diagrama de Arquitetura de Agentes, onde são representados os componentes arquiteturais dos agentes.

Os componentes internos dos agentes são representados com base nas conversações criadas no passo anterior, bem como os atributos de cada agente. No mínimo, cada ação criada no Diagrama de Comunicação deve ser definido como uma operação na montagem do Diagrama de Arquitetura de Agentes

No Diagrama de Arquitetura de Agentes, a linha tracejada significa interação com sistemas externos, enquanto a outra significa interação com outros agentes do sistema. A imagem 2.10 possui a notação utilizada no Diagrama de Arquitetura de Agentes.

Design do Sistema

A fase final da metodologia MASE consiste na criação de um diagrama de *deploy* dos agentes. Este diagrama consiste na representação do número de agentes que serão criados por cada máquina da aplicação, bem como suas localizações no sistema.

A imagem 2.11 contém um exemplo de diagrama de deploy. As caixas de três dimensões representam os agentes e as linhas representam as interações. Os retângulos com linhas tracejadas representam os ambientes que podem existir no SMA.

2.5 Java Agent Development Framework - JADE

JADE é um *middleware* desenvolvido em 1988 pela Telecom Italia, posteriormente (2000) tornando-se em um projeto open source. Seu desenvolvimento de baixo acoplamento permite que seja possível integrar várias bibliotecas auxiliares (*addons*) para facilitar o desenvolvimento de aplicações.

De acordo com [8], ele foi desenvolvido seguindo todas as especificações da FIPA, o que garante uma intercomunicação com outras plataformas. O JADE foi desenvolvido na linguagem JAVA, possibilitando o uso de diversas bibliotecas e frameworks desenvolvidos na linguagem.

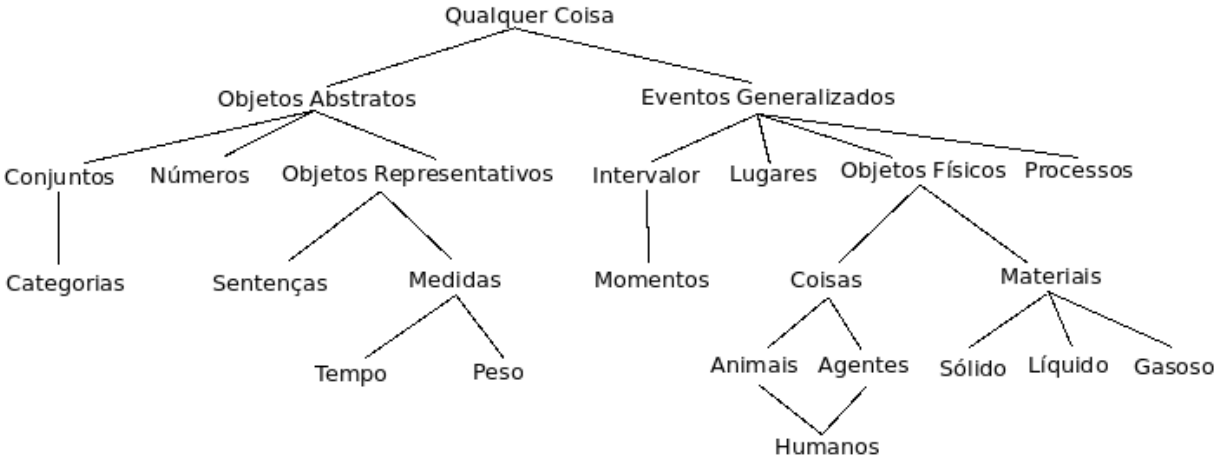

Figura 2.11: Notação utilizada no diagrama de deploy.

O middleware provê várias funcionalidades básicas que abstraem e simplificam o desenvolvimento de aplicações, com o objetivo do desenvolvedor estar mais preocupado com o comportamento do agente do que com a infra estrutura da plataforma.

O sistema de mensagem no JADE funciona de forma assíncrona. Um agente não precisa estar necessariamente disponível para receber as mensagens, visto que elas são enfileiradas e processadas em ordem. Além disso, não é necessário que um agente tenha uma referência para outro agente a fim de comunicar-se.

A arquitetura de um SMA desenvolvido em JADE funciona de forma semelhante à rede P2P (*Peer-to-Peer*), onde cada agente possui um nome único na plataforma - Agent ID (AID) - e é livre para entrar e sair a qualquer momento durante a execução. Uma plafatorma JADE possui normalmente os seguintes elementos:

- Agent Management System (AMS) é o agente responsável por supervisionar toda a plataforma e criar um elo entre os agentes. Esse tipo de serviço é chamado de *white pages* e indexa todos os agentes da plataforma.
- Directory Facilitator (DF) é o agente responsável por registrar todos os serviços e prover a funcionalidade de busca para os agentes. Este tipo de serviço é chamado de *yellow pages*.

Os agentes executam em threads separadas garantindo o não compartilhamento de recursos para evitar condições de corrida. A plataforma é responsável também por manter o ciclo de vida dos agentes. Durante a criação dos agentes, eles são automaticamente registrados no serviço de *white pages*

A mobilidade de agentes entre máquinas também é feita de forma transparente pelo JADE. De acordo com [8], a mobilidade do agente pode transportar o estado do agente (sob certas condições) entre processos e máquinas.

Uma das características mais importantes do *middleware* é o suporte nativo à ontologias e linguagens de comunicação FIPA. As ontologias podem ser modeladas (expressando ações, conceitos e predicados) de forma simples e clara, além de ser possível restringir as ontologias à determinados agentes que devem a conhecer.

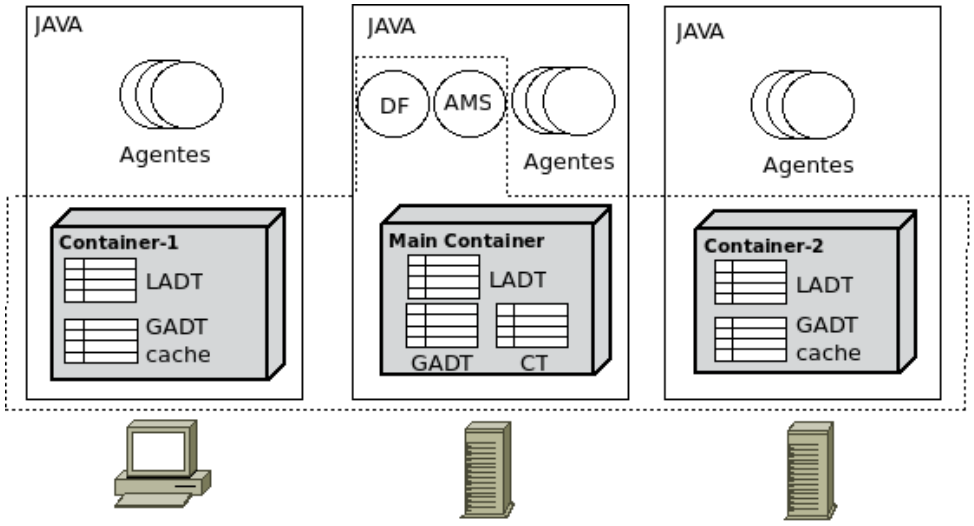

Figura 2.12: Representação da arquitetura principal do JADE.Fonte [8].

2.5.1 Arquitetura

A imagem 2.12 representa a arquitetura principal do funcionamento do JADE. Nela, é possível observar os principais elementos da arquitetura JADE distribuídos em três máquinas (dois servidores e um *desktop*). Cada máquina possui um container onde vários agentes residem.

O container principal(*main container*), presente na máquina do centro, possui algumas diferenças entre os outros, chamados de *Container-1* e *Container-2*, abringando agentes primordiais na plataforma que atendem às especificações da FIPA (número 23 e 61 [7]). O container principal possui o componente *container table* (CT), que possui as referências para os objetos e os endereços de comunicação dos nós que compõe a plataforma.

Além disso, o container principal possui uma tabela global para descrição de agentes (GADT) que registra todos os agentes da plataforma e o seu endereço. Os outros containeres possuem uma cópia em cache da tabela para, caso a tabela principal seja corrompida, possa ser substituída.

O primeiro deles é o *Directory Facilitator* (DF), agente responsável por registrar todos os serviços disponíveis na plataforma. O segundo agente é o *Agent Management System* (AMS) é o agente responsável por supervisionar toda a plataforma, registrando os agentes que estão rodando, bem como o seu estado.

2.5.2 Implementação dos Agentes

O JADE define a classe abstrata *Agent*, que é a base para todos os agentes definidos. O desenvolvedor tem apenas o trabalho de extender esta classe e implementar o comportamento no método *setup()*. O fato de extender a classe abstrata implica em herdar várias características já definidas pelo JADE (registro, configuração, etc.) e métodos que podem ser chamados para a implementação do comportamento do agente.

Cada agente possui um identificador único (Agent ID - AID) que identifica o agente em toda plataforma. Por padrão, o formato do AID possui primeiramente o nome do agente seguido do caracter '@', por fim o endereço da plataforma onde o agente está. Este AID é atribuído durante o registro do agente no AMS. Neste registro, é possível também registrar os serviços do agente no DF.

Conforme dito anteriormente, o método *setup()* deverá ser implementado e, no mínimo, deverá ser estabelecido um comportamento para o agente. Este comportamento diz respeito à ação que será realizada pelo agente durante a ocorrência de um evento. O registro/cancelamento dos comportamentos é feito pelos métodos exibidos no trecho 2.7.

Listing 2.7: Exemplo de registro de comportamento nos agentes.

```
void addBehaviour( Comportamento )
void removeBehaviour( Comportamento )
void addSubBehaviour( Comportamento )
void removeSubBehaviour( Comportamento )
```

Os comportamentos são separados em primitivos e compostos. A diferença entre ambos é a possibilidade dos comportamentos compostos poderem agregar vários outros comportamentos simples ou compostos, sendo eles: *ParallelBehaviour, SequentialBehaviour*. De maneira distinta, os comportamentos primitivos tem relação direta com o tempo, acontecendo durante um período de espera ou após o envio de uma mensagem. São eles: *SimpleBehaviour, CyclicBehaviour, TickerBehaviour, OneShotBehaviour, WakerBehaviour* e *ReceiverBehaviour*.

2.5.3 Ciclo de Vida dos Agentes

O JADE implementa o ciclo de vida especificado pela FIPA, podendo variar entre vários agentes de forma simples. Os seguintes estados são possíveis na plataforma:

- INITIATED - Após a criação do objeto, antes do registro do objeto no AMS, o agente assume o estado de iniciado. Este estado significa que o agente ainda não está disponível para a execução de ações na plataforma.
- ACTIVE - Neste estado o agente é registrado no AMS, possuindo assim o AID e o endereço. Ele está pronto para a execução do trabalho na plataforma.
- SUSPENDED - O agente está com as atividades suspensas e está em modo ocioso.
- WAITING - O agente está bloqueado esperando algum evento acontecer para executar alguma ação. Tipicamente, este estado é usado para fazer o agente esperar por alguma mensagem.
- DELETED - O agente está destruído e sua thread de execução é terminada. O seu registro será removido do AMS e a sua referência removida da JVM.
- TRANSIT - O agente está movendo-se de uma plataforma para uma nova localização. Mesmo em transito, é possível enviar mensagens para este agente, visto que serão empilhadas na sua fila de mensagens e posteriormente processadas quando ele assumir o estado ACTIVE.

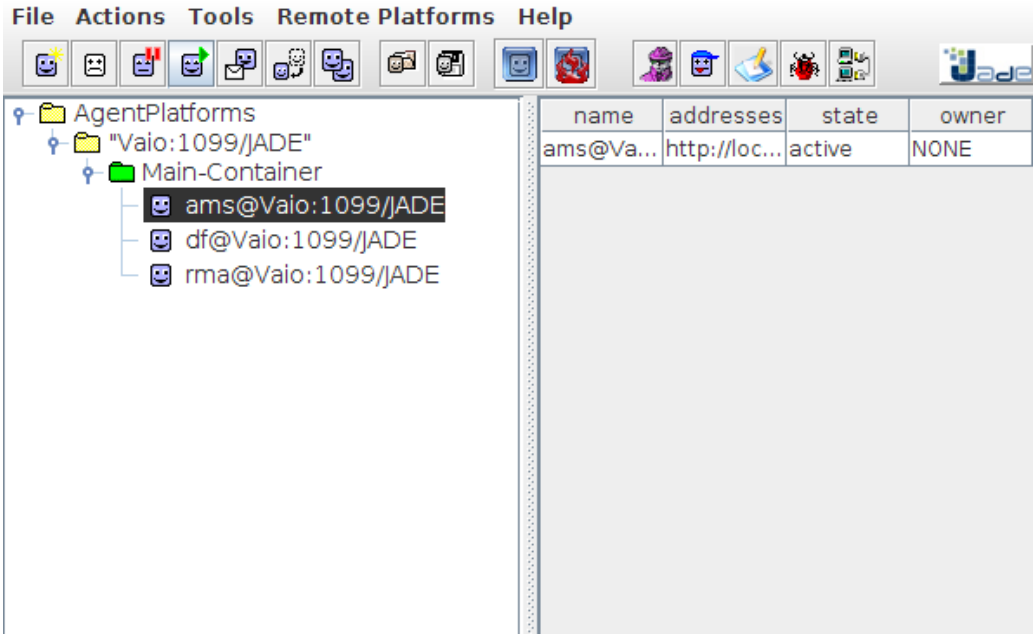

Figura 2.13: Apresentação da Interface

Para cada uma das transições de estados existem métodos que são invocados em um momento anterior. Eles são úteis para a execução de ações que antecedem a mudança de estado. Por exemplo: Durante a mudança do estado ACTIVE para DELETE, o método *doDelete()* é encarregado de implementar ações que antecedam o fim do agente, como que o agente desfaça o registro dos seus serviços no DF.

2.5.4 Interface Gráfica

O JADE permite o desenvolvimento de agentes com suporte à interface gráfica. Dessa forma, é possível desenvolver uma interação simples do agente com o ser humano.

Por padrão, o JADE utiliza diversos agentes que utilizam-se de interfaces gráficas para a comunicação dos humanos que, dentre outras funcionalidades, permitem o envio de dados, controle do agente e testes da plataforma. Ferramentas como gerência dos *containers*, visualização do DF, criação de mensagens a partir de agentes falsos (*sniffers*), dentre outros são disponibilizados nativamente para o desenvolvedor.

A imagem 2.13 apresenta o exemplo básico do agente RMA, onde é apresentado sua interface gráfica com a arquitetura em tempo real.

2.6 JBoss Seam

No mundo corporativo do JAVA, muitos frameworks são responsáveis por partes específicas de uma aplicação. Seguindo as especificações propostas para a plataforma (JSR), as aplicações implementam integrações com o banco de dados (Hibernate, JPA), integração com a camada de visualização (Struts 1 e 2, JSF), injeção de dependência (Weld e Spring). Porém a integração destes frameworks nem sempre é trivial, demandando muito tempo dos desenvolvedores a correta configuração.

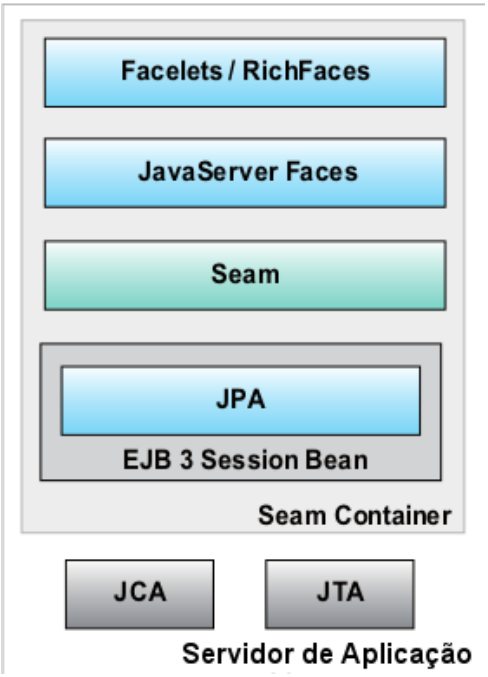

Figura 2.14: Representação da pilha de aplicações do Seam. [5]

Neste aspecto surge o JBoss Seam. Ele é um *framework* que reune as principais tecnologias de desenvolvimento web na linguagem Java. Ele integra as tecnologias *Asynchronous JavaScript and XML* (AJAX), *JavaServer Faces* (JSF), *Java Persistence* (JPA), *Enterprise Java Beans* (EJB 3.0) e *Business Process Management* (BPM) em uma única ferramenta que objetiva o desenvolvimento ágil de aplicações e o foco do programador na lógica de negócio [4].

A imagem 2.14 representa a pilha de aplicações do Seam, que são todos os *frameworks* utilizados pelo JBoss Seam.

Caso o desenvolvedor deseje utilizar outras tecnologias, o Seam provê configurações para que outras ferramentas possam ser integradas facilmente à aplicação.

De acordo com [5], uma das principais ferramentas do Seam é o *seam-generator*. Com ele, é possível gerar uma estrutura básica de projeto, com arquivos de construção, bibliotecas compatíveis e as configurações necessárias para o início do desenvolvimento e o *deploy* em um servidor de aplicação.

Além disso, uma das grandes vantagens do *seam-generator* é a geração automática de código, criando a partir de uma tabela no banco de dados todas as operações necessárias para a visualização, inserção, exclusão e atualização de dados (CRUD), diminuindo assim o tempo de desenvolvimento de aplicações.

Capítulo 3

Metodologia de desenvolvimento

Para a realização deste trabalho, foi planejada uma metodologia de desenvolvimento na qual objetivou-se um estudo sobre um módulo da plataforma JAMA. Inicialmente, faz-se necessário um levantamento do estado da arte para levantar os principais trabalhos que são relacionados à área de dependabilidade em sistemas distribuídos, para a orientação desse trabalho.

Em seguida é necessário levantar os requisitos do JAMA, descrevendo e modelando detalhadamente o módulo escolhido para a análise. A modelagem será feita com base na linguagem *Unified Modeling Language* (UML). De acordo com [14], UML consegue expressar o significado semântico de um projeto de software à todos os *stakeholders* utilizando, na grande maioria dos casos, notação gráfica.

Após esse levantamento inicial, será necessário também levantar requisitos de dependabilidade. Como já visto anteriormente, ambientes distribuídos possuem diversas características de dependabilidade relativos à arquitetura. Serão analisados no trabalho características que talvez não sejam adequados para outras arquiteturas.

O próximo passo é a caracterização das falhas que um ambiente distribuído pode ter. Falhas como erro na transmissão de mensagens para outros pares, perda de mensagens na rede, segurança na autenticação dos pares, ataques de negação de serviços da rede, propagação de vírus, dentre outros, devem ser requisitos considerados na análise a ser feita neste trabalho.

Em seguida se dá o desenvolvimento de um modelo probabilístico de estados com base em cadeias de *Markov*, para determinar as operacões do sistema que tem algum tipo de relação com os requisitos de dependabilidade acima mencionados.

O próximo passo é a implementação do modelo através do sistema de checagem de modelo *PRISM*. Em seguida a modelagem para análise formal utilizando pctl (*probabilistic computational tree logic*). Com todos essas análises e modelagens, será possível enfim realizar uma análise de sensibilidade mais detalhada sobre o componente do JAMA em questão.

Após a análise, caso seja necessário, serão propostas melhorias visando diminuir o grau de dependência de alguns componentes com o módulo avaliado. A hipótese a ser considerada é que o JAMA possuirá um baixo nível de sensibilidade de componentes do sistema. Essa melhoria pode ser no nível de estudos ou mesmo uma implementação em nível de código.

Por fim, essa solução deverá ser testada e avaliada afim de garantir a sua melhoria em relação à versão anterior da plataforma antes de ser realiazado esse estudo.

Referências

[1] Fipa communicative act library specification. http://www.fipa.org/specs/fipa00037/index.html. Acessado em 01/02/2013. 20

[2] The knowledge sharing effort. http://www-ksl.stanford.edu/knowledge-sharing/papers/kse-overview.html. Acessado em 01/02/2013. 17

[3] Object management group. http://www.omg.org/. Acessado em 05/01/2013. 4

[4] The seam framework - next generation enterprise java development. http://www.seamframework.org/. Acessado em 10/02/2013. 33

[5] D. Allen. *Seam in action*. Manning, 2009. vi, 33

[6] John L. Austin. *How to do things with words*. Harvard U.P., Cambridge, Mass., 1962. 16

[7] F. Bellifemine, G. Caire, T. Trucco, and G. Rimassa. Jade programmer's guide. *university of Parma*, pages 200–2003, 2002. 30

[8] Fabio L. Bellifemine, Giovanni Caire, and Dominic Greenwood. *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. Wiley, April 2007. vi, 28, 29, 30

[9] A. Cockburn. *Writing effective use cases*. Agile software development series. Addison-Wesley, 2001. 5, 7

[10] Germana Menezes da Nóbrega. Frank: Re-utilização de abordagens e ferramentas da ie em um sistema multi-agente para identificação de estilos de aprendizagem. October 2011. 2

[11] E.H. Durfee, V.R. Lesser, and D.D. Corkill. Trends in cooperative distributed problem solving. *Knowledge and Data Engineering, IEEE Transactions on*, 1(1):63–83, 1989. 15

[12] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003. vi, 4, 6

[13] Michael Genesereth, Richard E. Fikes, Ronald Brachman, Thomas Gruber, Patrick Hayes, Reed Letsinger, Vladimir Lifschitz, Robert Macgregor, John Mccarthy, Peter Norvig, and Ramesh Patil. Knowledge interchange format version 3.0 reference manual, 1992. 17, 19

[14] C. LARMAN. *Utilizando UML e Padrões*. Bookman, 3 edition, 2008. vi, 5, 7, 8, 10, 34

[15] David Poole, Alan Mackworth, and Randy Goebel. *Computational Intelligence: A Logical Approach*. Oxford University Press, 1998. 10

[16] A. Preece. Knowledge query and manipulation language: A review. *University of Aberdeen, Aberdeen, Tech. Rep*, 1997. 17

[17] Stuart Russel and Peter Novig. *Artificial Intelligence - A Modern Approach*. Number 1. Pearson Education, Upper Saddle River, New Jersey 07458, 1995. 11, 13, 20

[18] A.D. SCOTT, F. Mark, and H.S. CLINT. Multiagent systems engineering. *International Journal of Software Engineering and Knowledge Engineering*, 11(03):231–258, 2001. 23, 24, 25, 26

[19] J.R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cam: [Verschiedene Aufl.]. Cambridge University Press, 1969. 16

[20] Carlos Vinícius Sarmento Silva. Agentes de mineração e sua aplicação no domínio de auditoria governamental. Acessado em 18/11/2011. Disponível em http://monografias.cic.unb.br/dspace/handle/123456789/318, Abril 2011. 14

[21] J. Verschueren and J.O. Östman. *Key Notions for Pragmatics*. Handbook of Pragmatics Highlights. John Benjamins Publishing Company, 2009. 16

[22] Michael Wooldridge. *An Introduction to MultiAgent Systems*. Number 1. John Wiley and Sons Ltd, Krst Sussex POI0 IlJD, England, 2004. 11, 12, 14, 17, 19