Marcílio Ferreira de Souza Júnior

Um Arcabouço baseado em Componentes para Engenharia de Ambientes em Sistemas Multiagentes Abertos

DISSERTAÇÃO DE MESTRADO

Universidade Federal de Alagoas
Programa de Pós-Graduação em Modelagem Computacional de Conhecimento

Maceió-AL
Outubro/2007

Marcílio Ferreira de Souza Júnior

Um Arcabouço baseado em Componentes para Engenharia de Ambientes em Sistemas Multiagentes Abertos

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre pelo Programa de Pós-Graduação em Modelagem Computacional de Conhecimento da Universidade Federal de Alagoas

Prof. Dr. Evandro de Barros Costa
Orientador
Instituto de Computação - UFAL

Prof. Dr. Ângelo Perkusich
Orientador
Departamento de Engenharia Elétrica - UFCG

Maceió, Outubro de 2007

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecária Responsável: Helena Cristina Pimentel do Vale

S725u Souza Júnior, Marcílio Ferreira de.
Um arcabouço baseado em componentes para engenharia de ambientes em sistema multiagentes abertos / Marcílio Ferreira de Souza Júnior,. – Maceió, 2007.
74 f. : il.

Orientador: Evandro de Barros Costa.
Co-Orientador: Angelo Perkusich.
Dissertação (mestrado em Modelagem Computacional de Conhecimento) – Universidade Federal de Alagoas. Instituto de Computação. Maceió, 2007.

Bibliografia: f. 67-74.

1. Engenharia de ambientes. 2. Sistema multiagente. 3. Engenharia de sotware. I. Título.

CDU: 004.89

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Modelagem Computacional de Conhecimento pelo Programa Multidisciplinar de Pós-Graduação em Modelagem Computacional de Conhecimento, da Universidade Federal de Alagoas, aprovada pela comissão examinadora que abaixo assina:

Prof. Dr. Evandro de Barros Costa
UFAL – Instituto de Computação
Orientador

Prof. Dr. Arturo Hernández-Domínguez
UFAL – Instituto de Computação
Examinador

Profa. Dra. María del Rosario Girardi Gutiérrez
UFMA – Departamento de Informática
Examinadora

Maceió, outubro de 2007.

Agradecimentos

Aos meus pais Marcílio e Leni, que investiram nos meus estudos e me apoiaram no prosseguimento da minha vida acadêmica em todas as suas etapas. Mais um degrau foi alcançado e sei que estão orgulhosos disso. Retribuo, dessa forma, a educação sólida que eles sempre procuraram me proporcionar, mesmo nos momentos mais difíceis.

Às minhas irmãs Sue e Cacá, que acompanharam os dias difíceis da elaboração deste trabalho e, de forma indireta, contribuíram com sua escrita.

Aos professores Evandro Costa e Hyggo Almeida, pela orientação dada e pela enorme compreensão que tiveram durante a elaboração deste trabalho, principalmente devido à minha dedicação parcial e aos meus inúmeros compromissos profissionais. Em especial ao amigo Hyggo, que antes mesmo do início do mestrado sempre me apoiou e me impulsionou academicamente com várias conversas, críticas, sugestões, encontros e elaboração de artigos me preparando para assumir este compromisso e abrindo novas oportunidades.

À amiga Mônica Ximenes, pela inspiração, incentivo e força que me foram passados durante todos os momentos. Serei eternamente grato pelas longas noites e dias de revisão dos textos, pelo companheirismo, pelo carinho, pelos conselhos e, especialmente, pelas nossas conversas que me fizeram crescer como pessoa e como profissional.

Ao amigo Sóstenes Gusmão, por me incentivar fortemente no momento em que eu ainda estava com dúvidas em assumir este desafio. Lembro-me, até hoje, as frases ditas nos intervalos das aulas.

À Camila e Márcio, da UFAL, que durante as atividades do grupo de pesquisa me ajudaram no projeto e implementação, contribuindo para a finalização deste trabalho.

Obrigado a todos que de alguma forma contribuíram com este trabalho!!!

Por fim, agradeço a Deus por ter-me permitido chegar até aqui e poder fazer estes sinceros agradecimentos.

Resumo

Sistemas Multiagentes (SMA) são considerados um alto nível de abstração para projeto e engenharia de sistemas complexos, tendo sido caracterizados por estruturas de organização e processos de coordenação cada vez mais articulados e dinâmicos. Geralmente, agentes cooperam e coexistem dentro de um ambiente. Há um consenso geral na comunidade de agentes que os ambientes são parte essencial dos SMA dinâmicos e abertos. Contudo, diante das características dinâmicas presentes em tais SMA, apenas a utilização do paradigma de agentes no desenvolvimento de software não garante a flexibilidade e escalabilidade do projeto frente a inevitáveis mudanças de requisitos do mesmo. Por sua vez, o desenvolvimento baseado em componentes tem sido apontado como promissor na construção de aplicações com maior capacidade de adaptação a mudanças nos seus requisitos. Este trabalho tem como objetivo o desenvolvimento de um arcabouço para engenharia de ambientes de SMA abertos baseado no conceito de composição dinâmica de software. O arcabouço é baseado em uma especificação que procura mapear os conceitos de agentes em componentes para garantir a flexibilidade e reutilização provida na abordagem de componentes. Agentes e recursos são utilizados para compor o software, componentes são utilizados para compor agentes, e objetos e aspectos são utilizados para implementar as características funcionais e não-funcionais dos componentes. Os resultados favoráveis da presente proposta foram verificados nos experimentos realizados em quatro estudos de casos.

Abstract

Multiagent systems (MAS) are considered a high level abstration for design and engineering of complex systems. Such systems are characterized by organization structures and coordination process more articulated and dynamic. Usually, agents cooperate and coexist in an environment. In addition, there is a general consense in the research community that an environment is an essential part of open and dynamic MAS. However, given the dynamic characteristics present in complex systems, only the use of the agent-based paradigm in the software development does not guarantee the flexibility and scalability of the project ahead of the inevitable changes on requirements. For this reason, the component-based development have been identified as promising in the building of applications with greater ability to adapt to the changes of its requirements. This work aims at developing a component-based framework for engineering open MAS enviroments. The framework is based on the concept of dynamic software composition and supported by a specification that demand mapping from agents concepts to components in order to ensure the flexibility and reusability provided in the component approach. In addition, i) agents and resources are used to compose the software, ii) components are used to compose agents, and iii) objects and aspects are used to develop the functional and non-functional components requirements. The favorable results of this proposal were checked in experiments developed in four case studies.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Lista de Algoritmos

Capítulo 1. Introdução

As constantes mudanças nas características das organizações, a crescente competitividade e até mesmo a globalização acarretam freqüentes alterações nos requisitos de negócio, tornando os sistemas de software inerentemente complexos, com fortes características de evolução. O desenvolvimento de tais sistemas deve se utilizar de abstrações de alto nível para gerenciar a complexidade, ainda objetivando escalabilidade e flexibilidade para dar suporte a mudanças não antecipadas.

A abordagem multiagentes tem sido apontada como uma nova e adequada proposta para engenharia de sistemas de softwares complexos. Sistemas Multiagentes (SMA) (WEISS, 2000) (WOOLDRIDGE, 2002) são formados por um conjunto de agentes com regras e objetivos específicos e que interagem entre si de acordo com protocolos estabelecidos em sua sociedade. Weyns e Holvoet (2006) explicam que uma aplicação SMA é definida como um conjunto de entidades de software autônomas (agentes) que são organizados em estruturas compartilhadas (o ambiente). Os agentes procuram alcançar seus objetivos agindo sobre o ambiente e interagindo uns com os outros. De acordo ainda com (WEYNS e HOLVOET, 2006), os SMA estão muito próximos das arquiteturas de softwares, pois estruturam o sistema como um número de entidades autônomas que interagem dentro de um ambiente com o intuito de implementar requisitos funcionais e de qualidade do sistema, ou seja, solucionar um problema.

Como explanado acima, geralmente, agentes cooperam e coexistem dentro de um ambiente. Há um consenso geral na comunidade de agentes que a caracterização da entidade ambiente é fundamental no projeto de um SMA. Entretanto, o papel do ambiente nos SMA vem evoluindo. Na literatura atual existem inúmeras definições diferentes para ambiente (RUSSEL e NORVIG, 1995) (RAO *et al*, 1992) (PARUNAK, 1997) (FERBER, 1999) (DEMAZEAU, 2003) (ODEL et al, 2002) que, algumas vezes, é entendido como a infra-estrutura de hardware ou software onde o SMA é implantado e executado, ou é reduzido a um simples sistema de transporte de mensagens, ou, ainda, é visto como o "mundo externo" ou como um mediador para coordenação dos agentes. De acordo com estas definições, um ambiente provê as condições na qual uma entidade, agente ou objeto existem. Contudo, muitos pesquisadores e engenheiros continuam sem dar importância à integração do ambiente aos modelos e ferramentas de SMA como uma abstração primária e muitas vezes minimizam suas responsabilidades. Segundo Weyns *et al* (2004), uma conseqüência disso é que todo o

potencial das aplicações e técnicas que podem ser desenvolvidas usando SMA pode estar sendo negligenciado.

Enquanto a noção de ambiente não é bem definida por boa parte da comunidade, sendo ainda tratado como parte implícita do SMA e quase sempre de maneira *ad hoc*, e tendo em vista que os ambientes possuem uma variedade de responsabilidades, Weyns *et al* (2007:2) propõe o ambiente como uma parte explícita do SMA e o define como uma abstração de primeira ordem, distinguindo claramente as responsabilidades dos agentes e do ambiente, separando seus interesses. Desta forma, propõe-se a melhorar o gerenciamento da complexidade na engenharia das aplicações e tornar o ambiente um bloco de construção que engenheiros de SMA podem usar criativamente nos projetos de sistemas. Como resultado, consegue-se melhores práticas de engenharia e abre-se perspectivas para explorar o ambiente nos projetos de SMA.

A relevância dos ambientes como abstrações de primeira ordem é particularmente aparente para SMA situados (WEYNS e HOLVOET, 2004:1) (WEYNS e HOLVOET, 2004:2) (WEYNS e HOLVOET, 2007:1) (WEYNS e HOLVOET, 2007:2). Sistemas multiagentes situados são caracterizados pela presença de uma estrutura espacial explícita na qual os agentes são colocados. Geralmente, são caracterizados pela percepção específica e mecanismos de interação baseados em propriedades contextuais, como as posições relativas dos agentes. Este tipo de SMA enfatiza a importância da arquitetura dos agentes e do ambiente e define características como: funcionalidades para gerenciar percepções dos agentes sob o ambiente, entrega de mensagens, tratamento de ações dos agentes e manutenção dos processos que gerenciam o estado do ambiente independente dos agentes. As interações dos agentes com o ambiente são regidas por leis. Tais leis podem representar restrições específicas do domínio, como por exemplo, limitações da banda da rede. Em SMA situados, o ambiente é uma abstração arquitetural explícita com responsabilidades específicas que diferem das dos agentes.

No contexto deste trabalho, definiu-se ambiente como uma abstração de primeira classe que provê as condições essenciais para os agentes e recursos existirem e que mediam tanto a interação entre os agentes quanto o acesso aos recursos. Sob o ponto de vista da engenharia de software, considera-se o ambiente como uma entidade que contém recursos, situados ou não, e acessíveis pelos agentes de acordo com restrições impostas pelo próprio ambiente. Em (RUSSEL e NORVIG, 1995), são consideradas as seguintes propriedades para os ambientes: acessível ou inacessível, que determina se um agente tem acesso parcial ou completo ao estado do ambiente; determinístico ou não-determinístico, que indica se o

próximo estágio do ambiente é ou não completamente determinado pelo estado atual e as ações realizadas pelos agentes; episódico ou não-episódico, que indica se as interações entre agentes e ambiente seguem episódios de percepção/ação ou não; estático ou dinâmico, que determina se o ambiente pode mudar durante a deliberação de uma ação de um agente; discreto ou contínuo, que determina se existe um número pré-definido de percepções e ações para ser executada pelos agentes ou não.

Contudo, diante das características dinâmicas presentes nos sistemas complexos, apenas a utilização do paradigma de agentes no desenvolvimento de software não garante a flexibilidade e escalabilidade do projeto frente a inevitáveis mudanças de requisitos do mesmo. A necessidade de se construir aplicações mais robustas acarreta mudanças freqüentes de requisitos e por isso as aplicações devem estar preparadas pra tais mudanças. O grande problema são as mudanças não antecipadas, ou seja, quando o software já está pronto e este não foi preparado para tal feito. Porém, tal adaptação deve exigir esforço e tempo dos desenvolvedores para modificar a arquitetura, o projeto e o código do software. Essas mudanças não antecipadas têm sido apontadas como principal problema da evolução do software, pois não é algo que se possa prever durante o projeto do software (ALMEIDA *et al*, 2006:2). O software deve por si só suportar a evolução, em qualquer parte, e assim deveria ficar implícito para o desenvolvedor mecanismos que permitiram tal evolução. Além disso, nos casos de sistemas que não podem ser interrompidos, esta evolução deve ser gerenciada em tempo de execução.

O desenvolvimento baseado em componentes (CRNKOVIC, 2001) (McCLURE, 2001) tem sido apontado como promissor na construção de aplicações com maior capacidade de adaptação a mudanças nos seus requisitos. Segundo (Almeida, 2005) e (Almeida *et al*, 2004), a necessidade em agregar flexibilidade aos paradigmas de composição de software aumenta na proporção em que as características dinâmicas das aplicações evoluem. Porém, tal adaptação deve exigir esforço e tempo de implementação mínimos para atender ao requisito de *time-to-market* a que se submetem atualmente os sistemas de informação corporativos. Sendo assim, mecanismos de composição dinâmica de componentes tornam-se indispensáveis à possibilidade de adaptação dos sistemas de software com o mínimo de impacto possível, inclusive em tempo de execução. É neste contexto que Almeida (2004) propõe mecanismos que garantem a composição dinâmica de softwares baseados em componentes e em agentes impulsionado pela ausência de tais mecanismos.

O projeto COMPOR de Almeida (2004) - pode ser acessado no sítio *http://www.compor.net* - define uma especificação de composição de componentes,

denominada CMS (*Component Model Specification*) (ALMEIDA *et al*, 2006:1) (ALMEIDA et al, 2006:2) (FERREIRA *et al*, 2004), que garante a inexistência de referências explícitas entre provedores de funcionalidades de uma aplicação. Esta característica promove um aumento significativo na flexibilidade para inserir, remover e alterar tais provedores, até mesmo em tempo de execução. Também, uma especificação de um modelo de composição de agentes é definida no contexto do mesmo projeto. Esta especificação, denominada AMS (*Agent Model Specification*) (ALMEIDA, 2005) (NUNES *et al*, 2007)**,** procura definir entidades de abstração de composição orientada a agentes usando como base a CMS. A AMS procura mapear os conceitos de agentes em componentes para garantir a flexibilidade e reutilização provida na abordagem de componentes. Agentes e recursos são utilizados para compor o software, componentes são utilizados para compor agentes e objetos e aspectos são utilizados para implementar as características funcionais e não-funcionais dos componentes (ALMEIDA, 2005).

Este trabalho tem como ponto de partida os resultados teóricos e pragmáticos obtidos em (ALMEIDA, 2004) e (ALMEIDA, 2005) introduzindo uma infra-estrutura para engenharia de ambientes para sistemas multiagentes com foco na flexibilização do projeto, a qual considera todas as características de um ambiente multiagente, principalmente dinamicidade e ser aberto. Tal infra-estrutura é baseada na especificação AMS que por sua vez baseia-se na CMS, a qual promove evolução não-antecipada e dinâmica de software.

1.1 Problemática

Uma vez que as mudanças de requisitos tornaram-se freqüentes nas aplicações atuais, ter suporte à evolução de software minimizando o tempo e o custo tornou-se primordial. Porém, quando a evolução do software não pode ser antecipada, ou seja, após o término do software, o impacto sobre o projeto aumenta. O software deve por si só suportar a evolução, em qualquer parte, e assim devem ficar implícitos para o desenvolvedor que mecanismos permitiram tal evolução. Sendo assim, mecanismos de composição dinâmica são indispensáveis à possibilidade de adaptação dos sistemas de software com o mínimo de impacto possível, inclusive em tempo de execução, tais como sistemas de bancos, lojas, hospitais, dentre outros.

Analisando-se as propriedades dos ambientes em SMA encontradas em (RUSSEL e NORVIG, 1995) conclui-se que os mais complexos para os agentes são os do tipo inacessíveis, não determinísticos, não-episódicos, dinâmicos e contínuos. Ambientes com

estas propriedades são designados como abertos. Embora estes sejam os mais complexos, verifica-se que os ambientes físicos do mundo real são intrinsecamente abertos. Assim sendo, numa perspectiva de futuro desenvolvimento de agentes para operarem no mundo real, é importante definir projetos que permitam o desenvolvimento de agentes para operarem em domínios complexos ou, também designados, por ambientes abertos.

A principal característica de ambientes abertos e dinâmicos é que os agentes não são projetados para compartilhar um objetivo comum, além de possivelmente desenvolvidos por pessoas diferentes para atingir objetivos diferentes. Além disso, a composição do sistema pode variar dinamicamente enquanto os agentes incorporam e saem do sistema. Os recursos pertencentes a um ambiente aberto também podem ser redefinidos, assim como novos podem ser adicionados.

Levando em conta estes possíveis cenários, torna-se necessário flexibilizar o projeto do ambiente do SMA a fim de que os acoplamentos entre agentes e ambiente, agentes e recursos e entre recursos seja diminuído, ou até mesmo, inexista, de forma que as mudanças não antecipadas não acarretem alterações no software desenvolvido. Embora existam algumas abordagens para engenharia de ambientes para SMA, elas não cobrem todas as características dinâmicas impostas pelos ambientes abertos e podem ser usadas somente para domínios específicos, tal como será discutido no Capítulo 3 desta dissertação.

1.2 Objetivo do Trabalho

Este trabalho tem como objetivo o desenvolvimento de um arcabouço para engenharia de ambientes de SMA baseado no conceito de composição dinâmica de software. A infra-estrutura é a implementação da especificação AMS (NUNES *et al*, 2007) e busca prover suporte ao desenvolvimento e evolução de softwares orientados a agentes focando no nível do ambiente, incluindo os que possuam características dinâmicas e abertas.

São objetivos específicos:

- Aprimorar a AMS quanto aos conceitos de ambientes de SMA e realimentá-lo com novas características, atualizando as existentes;
- Implementar em Java o arcabouço JAF (*Java Agent Framework*). O projeto do arcabouço tem como base o arcabouço JCF (*Java Component Framework*) (ALMEIDA *et al*, 2006:2), que é a implementação da especificação CMS em Java.

- Definir quatro estudos de caso e implementar dois deles utilizando o arcabouço com o intuito de testar e buscar uma primeira avaliação para o mesmo.

Portanto, este trabalho se insere no contexto do Projeto COMPOR, mais especificamente como suporte ao sub-projeto COMPOR-*Frameworks*.

1.3 Relevância

Apesar de existirem algumas abordagens para engenharia de ambientes em sistemas multiagentes, elas não cobrem todas as características de um ambiente e podem somente ser utilizadas em um domínio específico. De uma maneira geral, as abordagens não consideram todas as características da classe geral e mais complexa de ambientes: a aberta. Ambientes abertos são classificados como inacessíveis, não-determinísticos, dinâmicos e contínuos.

Embora tenha havido uma proliferação de implementações de plataformas para SMA, os diversos arcabouços de desenvolvimento existentes (AVANCINI e AMANDI, 2000), (BIANCHINI *et al*, 2002), (BELLIFEMINE *et al*, 1999), (CHAUDHAN, 1997), (GRAHAM e DECKER, 1999), (GOLDSMITH *et al*, 1998), (MARTIN *et al*, 1999), (ZUNINO e AMANDI, 2000), (KENDALL *et al*, 2000) não oferecem suporte à composição dinâmica e evolução não antecipada do software orientado a agentes, nem tratam o ambiente de forma explícita como uma entidade separada.

Ademais, não existe uma implementação da especificação AMS. A construção de um arcabouço de software torna-se necessária para aplicar e validar os conceitos definidos na AMS. A implementação do arcabouço é uma contribuição esperada neste trabalho, de forma que, os conceitos do modelo de agentes para composição dinâmica de software tornem-se de fato uma ferramenta para os desenvolvedores de softwares orientados a agentes.

1.4 Estrutura da Dissertação

O restante deste trabalho está organizado como descrito a seguir:

- No Capítulo 2 é apresentada a fundamentação teórica sobre Ambientes de Sistemas Multiagentes, destacando-o como uma entidade de abstração de primeira ordem;
- No Capítulo 3 são apresentados os trabalhos correlatos, evidenciando as semelhanças e diferenças com o arcabouço proposto neste trabalho;

- No Capítulo 4, são apresentadas a CMS e a AMS como especificações que suportaram a implementação do arcabouço proposto;
- No Capítulo 5, são apresentados o projeto e a implementação do arcabouço JAF;
- No Capítulo 6, como forma de avaliar o arcabouço desenvolvido, são apresentados quatro estudos de casos para aplicações que envolvem diferentes tipos de ambientes, demonstrando a implementação de dois deles;
- Finalmente, no Capítulo 7 são apresentadas as conclusões do trabalho, apresentando as suas contribuições e discussões sobre trabalhos futuros.

Capítulo 2. Ambientes e Sistemas Multiagentes Abertos

Sistemas Multiagentes (SMA) são considerados um alto nível de abstração para projeto e engenharia de sistemas complexos, caracterizados por suas estruturas de organização e processos de coordenação que são cada vez mais articulados e dinâmicos (WEISS, 2000). Também, os SMA são considerados uma abordagem promissora para modelos e simulações destes mesmos tipos de sistema, como pode ser visto em (VALCKENAERS e HOLVOET, 2005), (WEYNS *et al,* 2005:1), (HAESEVOETS *et al*, 2007), (WEYNS e PARUNAK, 2007) e (STEINER *et al*, 2005).

De uma forma consensual em relação aos diversos domínios de aplicação - comércio eletrônico, recuperação de informação, redes de sensores, sistemas tutores inteligentes, entre outros - três entidades principais são identificadas no contexto da abordagem multiagentes: *Agente*, *Organização* e *Ambiente*. Estas entidades são definidas por diversos autores e em níveis de abstração diferentes. Além disso, de acordo com o domínio da aplicação novas características são atribuídas às mesmas.

Segundo Weyns e Holvoet (2004:2), há um consenso na comunidade de pesquisa em SMA que o *Ambiente* onde os agentes estão situados é uma parte essencial do sistema. Agentes e objetos de um SMA compartilham um ambiente comum. Tanto os agentes como os objetos estão dinamicamente inter-relacionados. É papel do ambiente definir as regras sob as quais os relacionamentos dos agentes podem existir, agindo assim como uma entidade estrutural para o SMA. Esta estrutura possui diferentes formas: pode ser espacial, organizacional ou pode ser uma entidade mediadora. A estrutura do ambiente é uma decisão de projeto que depende dos requisitos do domínio em questão, consistindo de uma propriedade fundamental do ambiente e deveria ser tratada de maneira explícita pelos engenheiros e projetistas do sistema como apresentado em (WEYNS *et al*, 2005:2), (WEYNS *et al*, 2005:3), (WEYNS *et al*, 2006) e (WEYNS *et al*, 2007:1).

Além dos agentes, um ambiente tipicamente compreende diferentes tipos de objetos ou recursos, situados ou não, e acessíveis pelos agentes de acordo com restrições impostas pelo próprio ambiente. É responsabilidade do ambiente controlar o acesso aos recursos. Geralmente, recursos podem ser lidos/percebidos, escritos/modificados ou consumidos pelos agentes. O alcance dos agentes para acessar um recurso particular depende de diversos

fatores, tais como: a natureza do próprio recurso, o relacionamento do recurso atual com os outros recursos ou agentes, a proximidade do agente ao recurso, as capacidades do agente, etc. De uma forma geral, os acessos aos recursos podem ser descritos por um conjunto de leis definidas pelo domínio da aplicação (WEYNS *et al*, 2004). A Figura 1 ilustra a perspectiva de um ambiente em um SMA destacando a ação e a percepção do agente sobre os recursos disposto nos ambiente.

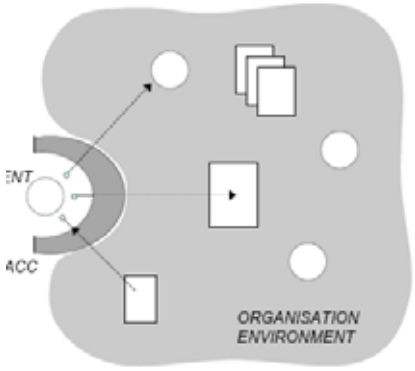

Figura 1- Perspectiva de um ambiente de SMA. Adaptado de (WEYNS et al, 2007:1)

Muitas vezes os agentes produzem algumas atividades no ambiente, como: comunicação, ação e percepção. Em um SMA os agentes precisam "conversar" entre si fazendo com que a comunicação seja um requisito imprescindível. Ela pode ser realizada de diferentes maneiras, variando desde uma transferência direta de mensagens ou indiretamente via um espaço compartilhado. Cada um destes tipos tem seus prós e contras, pois é uma decisão que depende do problema envolvido (WEYNS *et al*, 2004). Os agentes também realizam ações sobre o ambiente, modificando o seu estado para manipular algum recurso existente. As ações podem ser seqüenciais ou simultâneas. Gerenciar estas ações, no geral, é um problema complexo, pois necessita tratar os efeitos das mesmas para não haver violações no estado do ambiente. Além de agirem sobre o ambiente, os agentes precisam percebê-lo, implicando que o mesmo tenha a propriedade de ser observável, pelo menos parcialmente. Agentes inspecionam o ambiente de acordo com suas preferências. A percepção pode ser restringida pela capacidade do próprio agente, bem como pelas propriedades do ambiente que, de fato, refletem as propriedades do domínio do problema. Além das atividades que os agentes, recursos ou objetos podem produzir, o próprio ambiente possui processos que também pode produzir atividades nele mesmo. Um exemplo é um ambiente onde uma bola se move ou a temperatura local que evolui sobre o tempo.

Existem diferentes perspectivas nos papéis, os quais os ambientes podem exercer em um SMA e este aspecto vem evoluindo com o passar do tempo. Weyns *et al* (2007:2) explica

que alguns pesquisadores e desenvolvedores consideram o ambiente como externo ao sistema, isto é, o ambiente não faz parte explicitamente dos modelos ou arquiteturas, um exemplo clássico são os sistemas robóticos. Outro papel que o ambiente pode exercer é como um meio para coordenação de agentes, ou seja, em alguns tipos de sistemas, onde os agentes trabalham juntos para realizar seus objetivos, eles podem usar o ambiente para compartilhar informação e coordenar seu comportamento. Ainda, o ambiente pode exercer um papel de arquitetura provendo funcionalidades como gerenciamento de percepção, entrega de mensagens, manipulação de ações e manutenção do processo que gerencia o estado do próprio ambiente independente dos agentes. Por fim, ainda segundo (WEYNS *et al*, 2007:2), o ambiente pode assumir o papel de uma camada organizacional do SMA impondo um conjunto de restrições no comportamento dos agentes e oferecendo uma série de facilidades e serviços que os agentes podem usar.

Diversos modelos e definições para ambientes são propostos na literatura, cada um tratando de uma forma particular os conceitos e funcionalidades relacionadas com as interações entre agentes, e entre agente e ambiente. A seção a seguir apresenta uma revisão geral de definições representativas sobre ambientes e aponta a escolhida para este trabalho.

2.1 Definições de ambiente

Ao tratar a questão de ambientes de SMA, os pesquisadores possuem diferentes visões dessa noção, o que algumas vezes causa confusão. Tal confusão é agravada pelo fato da palavra "ambiente" também ser usada para referenciar a infra-estrutura de software ou de máquinas que o SMA é executado. Observando isto, Weyns e Holvoet (2004:2) identificam uma lista inicial de características básicas que os ambientes deveriam possuir:

- Independência: o ambiente é uma entidade ativa capaz de manter seu próprio estado, independente dos agentes que o compõem;
- Orientação a objetivos: o ambiente fornece serviços de suporte para os agentes prosseguirem para alcançar seus objetivos;
- Coordenação: o ambiente habilita os agentes para coordenar suas interações;
- Restrições globais: o ambiente reforça as restrições (leis) de todos os agentes do SMA;
- *Unboundedness*: enquanto um agente é um processo limitado, localizado no ambiente, o ambiente é extensível e ilimitado. Ambientes abertos permitem que agentes entrem e saiam à vontade;
- Observável: um ambiente de SMA pode ser observado, inspecionado e manipulado.

A literatura atual de ambientes possui um número substancial de importantes pesquisas, onde são propostos diversos modelos representativos. Weyns *et al* (2004) disponibiliza um *survey* do estado da arte das pesquisas relacionadas a ambientes de SMA. A seguir cada uma das definições de ambiente encontradas no referido *survey* são apresentadas.

Russel e Norvig (1995), no capítulo 2 de seu livro, discutem como um agente inteligente relaciona-se com seu ambiente: "Um agente é qualquer entidade que pode perceber o seu ambiente através de sensores e agir sobre o ambiente através de atuadores". Russel e Norvig (1995) também definem um ambiente como um programa genérico. Este simples programa oferece aos agentes percepções e recebe de volta suas ações. O programa então atualiza o estado do ambiente baseado nas ações dos agentes e possivelmente a partir de outros processos dinâmicos do próprio ambiente que não são considerados como sendo dos agentes. O programa de Russel e Norvig (1995) ilustra o relacionamento básico entre agentes e seu ambiente.

Em Rao *et al* (1992), são especificadas características do ambiente para uma larga classe de domínios de aplicação de sistemas baseados em agentes: a qualquer momento há potencialmente muitas formas diferentes que o ambiente pode evoluir; também a qualquer momento há potencialmente muitas formas diferentes de ações possíveis sob o ambiente; diferentes objetivos podem não ser simultaneamente armazenados no ambiente; as ações que armazenam os diversos objetivos dos agentes são dependente do estado do ambiente (contexto); o ambiente somente pode ser detectado localmente; a taxa em que computações e as ações podem ser realizadas está dentro dos limites razoáveis à taxa em que o ambiente evolui. Rao *et al* (1992) descrevem também as características típicas do mundo externo ao qual o sistema de agentes é implantado e com o qual o sistema interage.

Parunak (1997) define um ambiente como uma tupla {*Estado*, *Processo*}. O *Estado* é um conjunto de valores que definem completamente o ambiente, incluindo os agentes e recursos inseridos. O *Processo* indica que o próprio ambiente é uma entidade ativa. Ele possui seu próprio processo que pode mudar seu estado, independentemente de ações dos agentes. O principal propósito do *Processo* é implementar dinamismo no ambiente, como por exemplo um processo de manutenção de feromônios digitais. A definição de Parunak (1997) destaca a natureza ativa do ambiente.

Para Ferber (1999) o ambiente é definido como "um espaço *E* que geralmente tem um volume". Objetos, incluindo agentes, estão situados em *E*. Ou seja, em um dado momento, o objeto tem uma posição em *E*. Por sua vez, objetos são relacionados uns com os outros e os

agentes conseguem perceber estes objetos e manipulá-los. As ações dos agentes são sujeitas às "leis do universo" que determinam os efeitos das ações no ambiente. A essência do modelo do ambiente de Ferber (1999) é a maneira que ações são modeladas. Este modelo de ação faz distinção entre influências e reações. As influências partem dos agentes e são tentativas de modificar o curso dos eventos no "mundo". As reações, que resultam das mudanças do estado do ambiente, são produzidas pelo ambiente combinando as influências de todos os agentes, dado o estado local do ambiente e as "leis do universo". Esta clara distinção entre os produtos gerados do comportamento dos agentes e da reação do ambiente possibilitam uma forma segura de manipular atividades simultâneas no SMA. A definição de Ferber (199) destaca a função de container do ambiente e enfatiza a separação das ações dos agentes no ambiente das reações resultantes dessas ações.

Segundo Demazeau (2003), para os sistemas baseados em agentes são considerados quatro blocos de construção essenciais: agentes (as entidades que processam), interações (os elementos que estruturam as interações internas entre entidades), organizações (elementos que estruturam conjuntos de entidades) e, finalmente, o ambiente que é definido como "um elemento dependente do domínio que estrutura interações externas entre as entidades". O ambiente na perspectiva de Demazeau enfatiza as qualidades estruturais dos elementos externos ao sistema de agentes.

Um trabalho clássico que trata a modelagem de ambientes para SMA encontra-se em (ODEL *et al*, 2002). De acordo com Odell *et al* (2002), "um ambiente provê as condições nas quais uma entidade (agente ou objeto) existe". Os autores fazem uma distinção entre o ambiente físico e o de comunicação. O físico fornece as leis, regras, restrições e políticas que governam e dá suporte a existência física dos agentes e objetos. Já o ambiente de comunicação fornece os princípios e processos que governam e suportam o intercâmbio de idéias, conhecimento e informação entre os agentes e, ainda, as funções e estruturas que são comumente empregadas para estabelecer uma comunicação, como papéis, grupos e protocolos de interação entre os papéis e os grupos. Odell *et al* (2002) também define um ambiente social dos agentes como "um ambiente de comunicação no qual os agentes interagem de uma maneira coordenada". O ambiente social consiste de grupos de agentes, seus papéis na interação social e todos os outros membros que possam fazer parte. A definição de ambiente de Odell *et al* (2002) destaca as diferentes estruturas do ambiente – físico, comunicativo e social – e sua natureza mediadora.

Recentemente, Weyns *et al* (2007:2), colocam a noção de ambiente como uma entidade de primeira ordem e o tratam de forma explícita nos SMA. Weyns *et al* (2007:2) discutem que as atuais práticas em SMA consideram o ambiente essencialmente como uma "infra-estrutura" para agentes. Segundo Weyns *et al* (2007:2), esta perspectiva não explora todo o potencial dos ambientes em SMA e sendo uma infra-estrutura, o ambiente tipicamente controla um conjunto de responsabilidades específicas do sistema. Isto dificulta a flexibilidade nas atribuições de papéis entre os agentes e o ambiente. Para uma aplicação particular, todas as responsabilidades que não são controladas pela infra-estrutura são delegadas aos agentes, fazendo com que estes sejam mais complexos do que o necessário. Além do mais, infra-estruturas são projetadas para um tipo particular de abordagem (feromônios, campos, normas, etc). Sendo assim, Weyns *et al* (2007:2) introduzem a definição de ambiente como "uma abstração de primeira ordem que fornece as condições essenciais para os agentes existirem e que media a interação entre agentes e ao acesso aos recursos". Desta forma, são apontadas algumas responsabilidades que podem ser atribuídas ao ambiente, a saber:

- *O ambiente estrutura o SMA*: o ambiente é o primeiro de todo o espaço compartilhado entre os agentes, recursos e serviços, estruturando dessa forma o sistema inteiro. Os recursos possuem um estado específico que podem ser manipulados pelos agentes e os serviços são considerados como entidades reativas que fornecem funcionalidades para os agentes. Os agentes – bem como os recursos e serviços – são dinamicamente inter-relacionados entre si. É papel do ambiente definir as regras para estas relações. Portanto o ambiente age como uma entidade que estrutura o SMA. De um modo geral, a estrutura pode ser física (espaço, topologia, etc), de comunicação (transferência de mensagens) ou social (em termos de papéis, grupos e sociedades);
- *O ambiente embute recursos e serviços:* uma funcionalidade importante do ambiente é embutir recursos e serviços. Estes são tipicamente situados em uma estrutura física. O ambiente deveria fornecer suporte em um nível de abstração que esconda detalhes de baixo nível dos recursos e serviços aos agentes;
- *O ambiente pode se manter dinâmico:* Além das atividades dos agentes, o ambiente pode ter seu próprio processamento, independente dos agentes. Um exemplo típico é a evaporação e difusão de feromônios digitais.
- *O ambiente é localmente observável pelos agentes:* ao contrário dos agentes, o ambiente deve ser observável. Agentes deveriam ser aptos a inspecionar as diferentes

estruturas do ambiente, como seus recursos, serviços e possivelmente estados externos de outros agentes, dependendo de suas tarefas e objetivos e a ainda com um correto nível de abstração sem precisar entender detalhes de baixo nível;

- *O ambiente é localmente acessível pelos agentes:* agentes devem conseguir acessar as diferentes estruturas do ambiente. Como a observabilidade, o acesso a estrutura é limitado pelo contexto atual que o agente foi programado. Similarmente a observabilidade também, os agentes deveriam conseguir acessar o ambiente a partir de um certo nível de abstração.
- *O ambiente pode definir regras para o SMA:* o ambiente pode definir diferentes tipos de regras para todas as entidades do SMA. As regras podem se referir a restrições impostas pelo domínio ou leis impostas pelos projetistas. As regras restringem o acesso a recursos e serviços específicos para um tipo particular de agente. Impondo regras as atividades dos agentes, o ambiente age como um árbitro que tenta preservar o SMA em um estado consistente de acordo com as propriedades e as exigências do domínio da aplicação.

Tal como já mencionado, um fator importante a considerar no projeto de um SMA é o tipo de ambiente no qual seus agentes e recursos estarão situados, determinando assim qual o tipo de representação de ambiente sobre a qual cada um dos agentes deverá trabalhar, bem como a maneira de atuar e de perceber as alterações no ambiente. Russel e Norvig (1995) discutem um número de propriedades chaves dos ambientes que são amplamente adotadas pelos pesquisadores:

- Acessível vs. Inacessível: um ambiente acessível é aquele onde um agente consegue obter, através dos seus sensores, informação atualizada, precisa e completa sobre o ambiente. Grande parte dos ambientes típicos não são acessíveis neste sentido. Entre os ambientes inacessíveis destacam-se a Internet e todos os ambientes físicos reais;
- Determinístico vs. Não determinístico: em um ambiente determinístico cada ação tem um efeito único garantido, não existindo qualquer incerteza quanto ao resultado da sua execução;
- Episódico vs. Não-Episódico: em um ambiente episódico a experiência do agente é dividida em episódios. Cada um deles consiste em percepções e ações dos agentes, e a qualidade de cada ação depende somente do episódio em si.

- Estático vs. Dinâmico: um ambiente estático é suposto permanecer inalterado enquanto o agente decide a próxima ação a executar. Em contraste, num ambiente dinâmico outros agentes encontram-se a agir no ambiente ao mesmo tempo. Todos os ambientes físicos do mundo real e a Internet são dinâmicos neste sentido;
- Discreto vs. Contínuo: um ambiente diz-se discreto se existe um número finito de percepções e ações possíveis para o agente e contínuo caso contrário. Um ambiente pode ser contínuo no que diz respeito às percepções do agente e discreto no que diz respeito às ações (e vice versa).

Conforme mencionado anteriormente, ao analisar as propriedades dos ambientes conclui-se que os mais complexos para os agentes são os do tipo inacessíveis, não determinísticos, não-episódicos, dinâmicos e contínuos. Ambientes com essas propriedades são rotulados de **ambientes abertos** (HUBNER, 1995). Como tal, e numa perspectiva de futuro desenvolvimento de agentes para operarem no mundo real, interessa ser capaz de definir metodologias e ferramentas que permitam o desenvolvimento de agentes para operarem em domínios complexos, ou também designados, por ambientes abertos. A principal característica destes tipos de ambientes é que os agentes entram e saem do sistema à vontade, de forma dinâmica. A mesma característica se estende para os recursos localizados no ambiente, pois estes também podem ser removidos, assim como novos podem ser adicionados.

2.2 Engenharia de Ambientes

A seção anterior apresentou uma ampla discussão sobre o assunto ambientes no contexto de SMA, destacando os seus diversos desafios de concepção dos modelos referenciais. Entretanto, é requerido que os benefícios em torno dos conceitos se concretizem em aplicações práticas para os problemas. Um dos desafios da *Engenharia de Software baseada em Agentes* (JENNINGS, 1999) (JENNINGS, 2001), é definir práticas de projeto e implementação para engenharia de ambientes.

Segundo Weyns *et al* (2004), sob o ponto de vista de projeto, as práticas disciplinadas para agentes em geral ainda estão muito imaturas e, estender estas técnicas aos ambientes, aumenta extremamente o escopo do trabalho que precisa ser feito. Portanto, uma primeira etapa é reconhecer o ambiente como entidade de primeira ordem. Vários pontos discutidos nas seções anteriores impactam diretamente no projeto de um ambiente, como por exemplo, a

diversidade de mapeamento de domínios para diferentes tipos de ambientes e como os agentes estão relacionados com ele. Weyns *et al* (2004) afirma que os resultados dos estudos na área de ambientes necessitam ser capturados em ferramentas e técnicas como linguagens descritivas e outros mecanismos de representação, que auxiliarão os engenheiros a se comunicarem sem ambigüidades a respeito das alternativas de *design*, inclusive tratando a integração com domínios que necessitam ser físicos, como na robótica, por exemplo.

O crescimento da programação orientada a agentes levou a uma proliferação de metodologias, arcabouços e plataformas de desenvolvimento para SMA. Porém, os arcabouços populares como Jade (BELLIFEMINE *et al,* 1999), Jack (HOWDEN *et al*, 2007), Retsina (SYCARA *et al*, 2003), JAFMAS (CHAUHAN, 1997), Impact (ROGERS, 1999), Sim Agent (SLOMAN e POLI, 1996) e Zeus (NWANA *et al*, 1999) reduzem o ambiente a um simples sistemas de transporte de mensagem ou a um *broker*. Ainda, metodologias reconhecidas como *Message* (EVANS *et al*, 2001), Prometheus (PADGHAM e WINIKO, 2002), MaSE (DELOACH, 2001), *Tropos* (BRESCIANI *et al*, 2004) e até mesmo FIPA (FIPA, 2007) oferecem suporte a alguns elementos básicos do ambiente até no máximo a etapa de análise, mas falhando em não tratá-lo como uma entidade primária e explícita no SMA e em nível de projeto de software. O aumento do reconhecimento da importância dos ambientes em SMA estimulou a expansão das funcionalidades destas ferramentas ou até mesmo influenciaram o desenvolvimento de novas que suportam mais amplamente os ambientes, além dos agentes.

Capítulo 3. Trabalhos Correlatos

A existência de diversas propostas de ferramentas ou *middlewares* para suportar aplicações multiagentes evidencia a exploração de soluções que possuam real uso em diferentes domínios de aplicação. Estas infra-estruturas tipicamente provêem serviços fundamentais para criação, gerência, descoberta e comunicação de agentes. Entre as infra-estruturas disponíveis, várias delas são *open source* e podem ser usadas livremente, outras são vendidas comercialmente e variam em preço e capacidade.

O presente trabalho, tal como já mencionado, propõe um arcabouço para engenharia de ambientes de sistemas multiagentes baseado no mecanismo de composição dinâmica de software, definindo um conjunto de conceitos com o objetivo de oferecer suporte à reconfiguração dinâmica de componentes, e por sua vez, de agentes e recursos. As características dinâmicas encontradas nos sistemas complexos, particularmente em SMA´s que atuam em ambientes abertos, procuram ser supridas pelo arcabouço através deste mecanismo.

Desta forma, tenta-se mapear os conceitos de agentes e ambientes em componentes para garantir a flexibilidade e reutilização provida em tal abordagem. Agentes e recursos são adotados para compor o software, componentes são utilizados para compor os agentes e recursos, e objetos são utilizados para implementar as características funcionais e não-funcionais dos componentes.

Este Capítulo apresenta uma discussão sobre as propostas mais relevantes de arcabouços e infra-estruturas para SMA e ao mesmo tempo procura destacar suas principais características no intuito de evidenciar as semelhanças e diferenças com o arcabouço proposto neste trabalho.

3.1 Retsina

Retsina (*Reusable Environment for Task-Structured Intelligent Network Agents*) (SYCARA *et al*, 2003) é uma infra-estrutura multicamadas para sistemas multiagentes abertos que suporta comunidades de agentes heterogêneos.

Retsina define a infra-estrutura do SMA como um conjunto de serviços, convenções e conhecimentos que suporta interações sociais complexas. - provê a plataforma na qual os componentes da infra-estrutura e os agentes executam. Retsina oferece suporte a uma

variedade de plataformas de execução e automaticamente gerencia os diferentes tipos de camadas de transporte de redes.

As camadas intermediárias do ambiente provêem funcionalidades de comunicação baseada em transferência de mensagens ou via *multicast* para descobrir processos que permitam os agentes encontrarem componentes na infra-estrutura. Além da comunicação, existe a camada de gerenciamento dos serviços que oferece suporte para monitorar as atividades dos agentes e as aplicações carregadas. Também provê um serviço para observar a performance de simulações no ambiente. O módulo de segurança oferece autenticação aos agentes, comunicação segura e integridade dos componentes da infra-estrutura.

No nível mais alto da arquitetura do Retsina encontra-se o serviço de nomes de agentes. O objetivo desta camada é abstrair as localizações físicas dos agentes mapeando identificadores para endereços de redes. Não há interferência nas transações entre agentes, os identificadores somente provêem aos agentes os endereços que eles podem armazenar informações em *cache*, removendo a necessidade de *lookups* desnecessários aos demais agentes localizados no ambiente.

Contudo, esta infra-estrutura não oferece características dinâmicas para agentes e recursos, em outras palavras, os recursos e agentes não conseguem entrar ou sair do ambiente dinamicamente.

3.2 DIVAS

Em (MILI *et al*, 2004), é apresentado um ambiente para Sistemas Agentes-Ambientes (AES). O ambiente DIVAS é responsável por manter um estado estável no qual os agentes existem. DIVAS é uma ferramenta de simulação baseada em agentes com o propósito de desenvolver, gerenciar e visualizar mundos artificiais sociais de larga escala. Ele oferece a idéia de separar o agente de suas responsabilidades no ambiente e imbuir o ambiente com mais responsabilidades, ambos o ambiente e os agentes são mais robustos e adaptáveis.

Em DIVAS, o mundo artificial consiste de um conjunto de agentes e um ambiente. Os agentes são móveis no sentido de que eles não são limitados a estar um único lugar podendo percorrer o ambiente de local em local a fim de alcançar seus objetivos. O ambiente é um mapa de localização constituído de pedaços (nós e células) e bordas (*pathway*). Ele também possui um conjunto d e objetos, que são entidades passivas que podem ser percebidas, modificadas e removidas pelos agentes. O objetivo do ambiente consiste em manter a rede de células estáveis para que possa ser provida aos agentes uma visão de seu estado atual.

O *framework* DIVAS possui quatro componentes principais. O *Agent Framework* cria mundos artificiais de larga escala. O *Data Management System* extrai as informações das estruturas emergentes usando técnicas de *data mining* e *clustering*. O *Visualization Framework* usa gráficos para desenhar algoritmos para visualizar a dinamicidade do número de agentes. O *Prediction System* gera modelos de predição quando os dados são esparsos e incompletos.

Mili *et al* (2004) define cinco perspectivas ortogonais para DIVAS: topológica, comportamental, dado, funcional e construcional. Uma clara distinção entre elas permite um melhor entendimento do problema. A topologia de um ambiente em DIVAS define a manipulação de sua estrutura como um grafo em uma visão 2D de localização. A perspectiva comportamental descreve o ambiente em termos de seus estados, transições e eventos. Mais precisamente ele representa o vínculo entre os eventos e a resposta do ambiente durante a execução. Os dados são armazenados no ambiente usando células. Cada célula contém informações da fronteira do "território", suas bordas e a informação em si.

Neste trabalho tanto os agentes como o ambiente possuem recursos próprios. Entretanto, os objetos dentro do ambiente não são consumidos pelos agentes, eles podem ser percebidos, modificados ou destruídos pelos agentes. Claramente há uma separação entre o ambiente e o agente e também permite construir ambientes situados.

3.3 ObjectPlaces

Schelfthout e Holvoet (2004) descrevem um sistema de *middleware* chamado *ObjectPlaces*. Esta proposta para ambiente é baseada na abordagem de *tuplespaces*, que é um possível mecanismo para oferecer uma coordenação nos ambientes para SMA através do compartilhamento de tuplas.

ObjectPlaces mantém uma visão definida para o agente representando o estado atualizado das *tuplespaces* dos nós vizinhos da rede de agentes e, ainda, esta representação é mantida quando a rede e o conteúdo das tuplas mudam. Esta facilidade pode ser realizada eficientemente porque a interface para a tupla no *ObjectPlaces* é assíncrona, isto é, as operações não bloqueiam as tuplas e seus resultados são retornados quando estão disponíveis.

Este ambiente para SMA satisfaz três requisitos principais: observável, ativo e dinâmico. O ambiente pode ser modificado sem manipulação direta de um agente, em outras palavras, ele muda naturalmente.

Diferente deste trabalho, onde os recursos podem ser lidos/percebidos, escritos/modificados ou consumidos pelos agentes, no *ObjectPlaces* os agentes invocam operações para adicionar e remover objetos ou para observar o conteúdo do *objectplace*. Logo, não é possível mudar uma propriedade de um recurso ou invocar determinado serviço disponibilizado pelos recursos. O *ObjectPlaces* permite construir ambientes dinâmicos e situados, porém não os trata como elemento de abstração de primeira ordem.

3.4 CartAgO

Ricci *et al* (2007) apresentam uma infra-estrutura chamada CartAgO (*Common "Artifacts for Agents" Open infrastructure*). Esta infra-estrutura trata os ambientes como parte primária de um SMA, diferente dos trabalhos discutidos anteriormente. Sua principal característica é o suporte direto a noção de artefato para engenharia de aplicações multiagentes. Ela provê serviços básicos para agentes instanciar e usar artefatos, bem como uma maneira flexível para engenheiros de SMA projetar e construir qualquer tipo útil de artefato. Artefatos são entidades reativas úteis, com um estado que pode mudar de acordo com as operações executadas pelos agentes.A abstração utilizando artefatos provê uma forma natural de modelar abstrações orientadas a objetos e serviços no nível de agentes, criando uma ponte entre as lacunas conceituais e semânticas do paradigma orientado a agentes.

O modelo abstrato de CartAgO compreende três partes básicas: o modelo adotado para caracterizar as ações e percepções relacionadas aos agentes e o seu ambiente computacional; o modelo adotado para os artefatos como elemento básico para engenharia do ambiente; e o modelo adotado para o *workspace* que é o contexto lógico onde os artefatos e agentes estão localizados. Devido ao uso de artefatos, o modelo de interação do CartAgO é diferente dos modelos geralmente adotados em infra-estruturas de agentes de software, que são tipicamente baseados somente em ações de comunicação. Ao invés disso, o modelo de interação de artefatos é mais similar aos modelos definidos para agentes situados e autônomos, no qual os agentes interagem com seu ambiente computacional por meio de ações apropriadas fornecidas pela criação/construção, seleção e uso dos artefatos e também pela percepção de eventos observáveis gerados pelos mesmos artefatos. Cada artefato possui um identificador lógico, especificado pelo criador do mesmo, que são utilizados como uma identidade única no ambiente. Eles também possuem uma interface de uso que os agentes podem explorar a fim de interagir com eles, ou seja, usá-los. A interface é definida como um conjunto de operações que podem ser invocadas observando-se os eventos gerados.

Em CartAgO, artefatos (e agentes) são logicamente colocados em *workspaces*, que podem ser usados para definir a topologia do ambiente computacional. Posteriormente, artefatos podem ser adicionados ou removidos dos *workspaces* e agentes podem dinamicamente entrar ou sair do *workspace*. Esta infra-estrutura utiliza a noção de artefato, enquanto a deste trabalho utiliza a abordagem de componentes.

Capítulo 4. Especificação do Modelo de Agentes

Na proporção em que as características dinâmicas das aplicações evoluem, a necessidade de agregar flexibilidade no desenvolvimento do software aumenta, fazendo com que os requisitos definidos em primeira instância de análise tendam a ser refinados ou alterados com a iteração do ciclo de vida. Assim, o desenvolvimento baseado em componentes (DBC) - como pode ser visto em (CRNKOVIC, 2001), (HEINEMAN e COUNCILL, 2001) e (LAU, 2004) - tem sido utilizado constantemente na produção de software com maior capacidade de adaptação a mudanças nos seus requisitos, promovendo aumento de produtividade e qualidade, em virtude da reutilização de componentes previamente implementados.

Heineman e Councill (2001) definem um componente de software como um artefato que obedece a um *modelo de componentes.* Esse modelo descreve como um componente deve ser visto externamente, podendo ser implantado e composto independentemente em um arcabouço, sem modificação, de acordo com um padrão de composição. No arcabouço são implementadas as especificações definidas no modelo de componentes. Esse modelo define, em um nível conceitual, as interações que regem os componentes e as entidades de software utilizadas para a construção de aplicações.

As abordagens do DBC geralmente são baseadas nos conceitos de modularidade, parametrização e conformidade de padrões. Nos sistemas que seguem esta abordagem, módulos são componentes com interfaces bem definidas que estabelecem um contrato a ser seguido pelos seus utilizadores. O conceito de parametrização é um fator determinante para a capacidade de adaptação do componente ao contexto no qual será reutilizado, permitindo que o componente venha a ser adaptado ao ambiente no qual executará. Quanto mais padronizado é um componente, maior a possibilidade de ser reutilizado. A conformidade pode ser referente a padrões estabelecidos por consórcios abertos ou por organizações neutras de padronização. Sendo assim, antes de desenvolver uma aplicação utilizando a abordagem de componentes, é necessário que sejam definidos o modelo e o arcabouço de componentes utilizados.

Em relação à abstração multiagentes, as características dinâmicas de sistemas explicitadas no início deste capítulo se refletem na alteração dos relacionamentos dos agentes e da entrada ou saída de agentes ou recursos do sistema. Muitos investimentos em Engenharia de Software objetivando prover ferramentas e metodologias para a utilização da abordagem multiagentes em larga escala têm sido realizados, como pode ser observado no capítulo

anterior. Porém, na maioria destes trabalhos, o impacto da alteração ou refinamento dos requisitos sobre o software não é levado em conta. Sendo assim, os relacionamentos entre os agentes do sistema não se alteram e novos agentes não podem passar a fazer parte do sistema ou, quando tais características são providas, isto não é refletido para as fases de projeto e implementação do software. Em resumo, o desenvolvimento do software não reflete a realidade de cenários evolutivos, cada vez mais presentes no contexto de engenharia de software (ALMEIDA, 2005).

Segundo Almeida (2005), para tratar desse problema, recentemente a pesquisa em agentes computacionais definiu novos paradigmas de programação com base em componentes autônomos que interagem para satisfazer aplicações adaptáveis e mutáveis. Além disso, sistemas mais escaláveis e flexíveis podem ser construídos em comparação com outros paradigmas, como o orientado a objetos, por exemplo. Portanto, ao invés de sistemas construídos com base na composição fixa e estrita de componentes definidas na fase de projeto, o objetivo é utilizar componentes autônomos, desacoplados e que interagem de forma transitória. Desta forma, sistemas de software abertos não precisam mais ser construídos por entidades com associações fixadas em tempo de projeto e sim, por entidades desacopladas e cujos relacionamentos podem mudar, inclusive, em tempo de execução.

As aplicações que lidam com ambientes multiagentes abertos, desenvolvidas com base na infra-estrutura de software apresentada neste trabalho, necessitam de suporte para a alteração em tempo de execução e para a evolução da aplicação sem um planejamento prévio, como por exemplo, a saída e entrada esporádica de agentes e recursos do ambiente. Essas duas características estão presentes no modelo de componentes do Projeto COMPOR (ALMEIDA, 2004), denominado CMS (*Component Model Specification*) (ALMEIDA *et al*, 2006:1).

De acordo com os problemas levantados acima, a CMS define um modelo de composição dinâmica de software baseado em componentes que suporta a evolução não-antecipada. A especificação CMS é a base para o Modelo de Agentes no qual o arcabouço para engenharia de ambientes abertos proposto neste trabalho está baseado. O modelo de componentes da CMS permite mudar qualquer parte do software construído, adicionando ou removendo componentes, até mesmo em tempo de execução. Esta característica é levada ao Modelo de Agentes, uma vez que este é uma extensão do primeiro. No intuito de embasar a implementação do arcabouço proposto, as próximas seções descrevem as especificações dos modelos de componentes e de agentes que lhe servem de suporte.

4.1 Especificação do Modelo de Componentes

Como mencionado anteriormente, o mecanismo de composição proposto busca maximizar o reuso e a flexibilidade do projeto do software. A especificação do modelo de composição de componentes CMS tem base no princípio de que a inexistência de referências explícitas, ou diretas, entre provedores de funcionalidades tem como conseqüência uma maior flexibilidade na inserção, remoção e alteração destes provedores, inclusive em tempo de execução.

De acordo com a CMS, um sistema baseado em componentes deve ser descrito como uma composição hierárquica de dois tipos de entidades: contêineres e componentes funcionais. Os componentes funcionais implementam as funcionalidades do sistema, disponibilizando-as em forma de serviços. Os componentes funcionais não são compostos por outros componentes, ou seja, não possuem "componentes-filhos". Os contêineres, por sua vez, não implementam funcionalidades, apenas gerenciam o acesso aos serviços e eventos dos seus "componentes-filhos", que podem ser componentes funcionais ou outros contêineres. Os componentes funcionais são disponibilizados através da inserção nos contêineres. É necessário que haja ao menos um contêiner (contêiner-raiz) para que seja possível adicionar os componentes funcionais. Cada contêiner possui uma lista dos serviços providos e eventos de interesse de cada um dos seus "componentes-filhos".

Após a inserção de um componente, a lista de serviços e eventos de cada contêiner até a raiz da hierarquia é atualizada. Isto é necessário para que os serviços providos pelo componente recém-adicionado estejam disponíveis para qualquer outro componente funcional do sistema. Além disso, permite que o componente adicionado seja notificado caso algum evento de seu interesse seja disparado por outro componente (ALMEIDA *et al*, 2006:2).

A especificação CMS permite que os componentes funcionais interajam através de dois mecanismos: modelo de interação baseado em *serviços* e modelo de interação baseado em *eventos*. No primeiro caso, nenhum componente pode invocar um serviço de outro, até mesmo quando este pertence a um outro contêiner. A interação baseada em eventos foca no anúncio de uma mudança de estado de um dado componente para todos os outros interessados. Em ambos os casos não há referência explícita entre eles.

O funcionamento do modelo de interação baseada em serviços encontra-se ilustrado na Figura 2 através de um exemplo. Assumindo a existência de um serviço chamado "save" implementado pelo componente *K* da aplicação, a execução deste serviço pode ser requisitada pelo componente *X*, sem uma referência explícita de *K*. Este processo é detalhado a seguir.

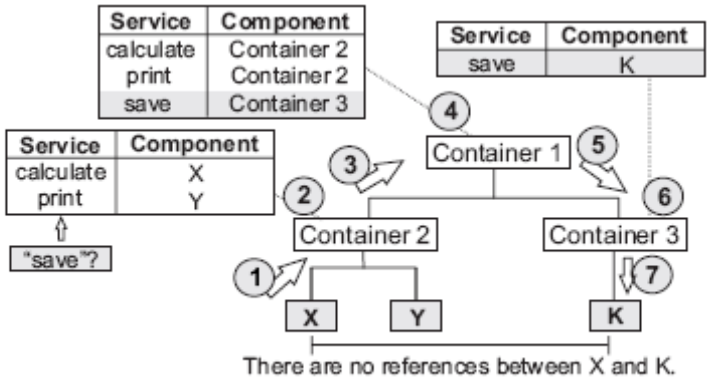

Figura 2 - Interação Baseada em Serviços da CMS (ALMEIDA et al, 2006:2)

1. O componente *X* requisita a execução do serviço "save" para seu contêiner pai;
2. Baseado na sua tabela de serviços, o *Contêiner 2* verifica que não há nenhum componente filho que implementa o serviço "save";
3. O *Contêiner 2* repassa a requisição para seu contêiner pai, neste caso o *Contêiner 1*;
4. O *Contêiner 1*, de acordo com sua tabela de serviços, verifica que um de seus filhos implementa o serviço "save", o *Contêiner 3* neste caso. O *Contêiner 1* visualiza o *Contêiner 3* como o componente que implementa o serviço requisitado;
5. O *Contêiner 1*então repassa a requisição ao serviço para o *Contêiner 3*;
6. O *Contêiner 3* não implementa o serviço, mas possui uma referência para o componente que implementa o serviço requisitado e repassa a requisição para ele, no caso o componente *K*;
7. O componente *K* executa o serviço "save" e retorna o resultado seguindo o caminho inverso, retornando ao componente requisitante.

É importante ressaltar que não há nenhuma referência entre o componente requisitante do serviço (*X*) e o componente que o provê (*K*). Portanto, é possível mudar o componente que fornece o serviço "save" sem modificar o restante da estrutura.

Por sua vez, o modelo de interação baseado em eventos da CMS é ilustrado na Figura 3 também através de um exemplo. Quando um evento é anunciado por um componente funcional, todos os da hierarquia da aplicação que estão interessados na ocorrência do evento devem ser notificados. A interação baseada em eventos também é implementada pelos contêineres e, portanto, não existe nenhuma referência entre os componentes funcionais envolvidos. Os passos desse processo são detalhados a seguir.

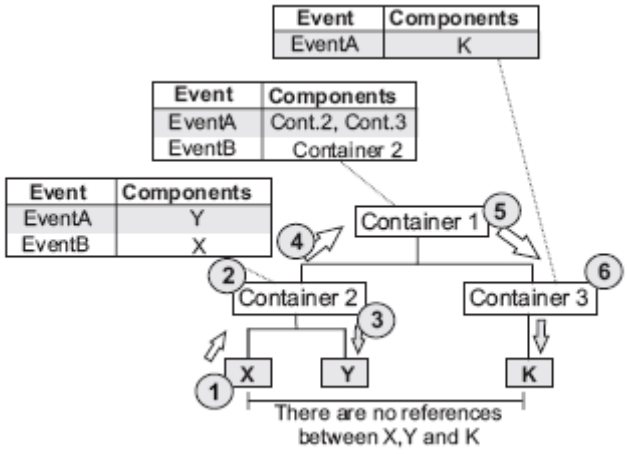

Figura 3 - Interação Baseada em Eventos da CMS (ALMEIDA et al, 2006:2)

1. O componente X anuncia um evento chamado "evento A";
2. O anúncio é diretamente recebido pelo seu contêiner pai (*Contêiner 2*), que verifica se algum de seus componentes-filhos deve ser notificado inspecionando sua tabela de eventos;
3. O *Contêiner 2* encaminha o evento para os componentes interessados, neste caso o C*omponente Y*;
4. O *Contêiner 2* repassa o evento para seu contêiner pai (*Contêiner 1*);
5. O *Contêiner 1*, de acordo com sua tabela de eventos, repassa o evento para todos aqueles interessados, exceto o que anunciou o evento (*Contêiner 2*). Como o *Contêiner 1* é a raiz da hierarquia, não existe nenhum contêiner pai para repassar o evento. Então o evento é somente encaminhado para o *Contêiner 3*;
6. O *Contêiner 3* repassa o evento de acordo com sua tabela de evento, que no caso somente o componente K é interessado.

Como pode ser visualizado na Figura 3, não há nenhuma referência entre o componente que anunciou o evento (*X*) e os componentes interessados no evento (*Y* e *K*). Logo, aquele que anunciou o evento pode ser trocado sem acarretar modificações no restante da estrutura.

Essa característica peculiar da CMS - a garantia que não existe dependência entre componentes funcionais – facilita a implementação de *composição dinâmica* nos arcabouços que seguem as especificações do modelo, ou seja, a tarefa de trocar componentes, inclusive em tempo de execução, é facilitada. Adicionalmente, a característica de suporte *a evolução de software não antecipada* está presente no modelo. Ao desenvolver aplicações com base na

CMS, não é necessário se preparar para possíveis mudanças que ocorrerão no software, ou seja, não é necessário antecipar as mudanças já que é permitido adicionar, remover ou alterar componentes até mesmo em tempo de execução. Essas duas características são cruciais na engenharia de ambientes abertos em SMA, como será apresentado mais adiante.

Além dos conceitos de *Contêineres* e *Componentes Funcionais*, a CMS define o conceito de *Adaptador*. Um *Adaptador* é um tipo especial de contêiner que possui uma referência para um único componente funcional no intuito de restringir o acesso para seus serviços e eventos. Um *Adaptador* pode ser criado, removido ou alterado em tempo de execução. A idéia por trás deste conceito é baseada no conhecido padrão de projeto *Adapter* (GAMMA *et a*l, 1995) (VLISSIDES *et al*, 1996).

4.2 Especificação do Modelo de Agentes

A composição dinâmica de componentes, integrada a objetos e a aspectos, é definida na especificação CMS apresentada na seção anterior. Esta abordagem possibilita a flexibilização do projeto de software e reutilização de componentes, e consiste na base para a abordagem de composição multiagentes definida na AMS (*Agent Model Specification*) (ALMEIDA, 2005) (NUNES *et al,* 2007), que suporta a implementação do arcabouço do presente trabalho.

Contudo, as entidades de abstração da abordagem multiagentes são diferentes daquelas da abordagem de componentes. Sendo assim, é necessário haver um mapeamento entre as duas para que se tenha na abordagem multiagentes a garantia de flexibilidade e reutilização provida na de componentes. Da mesma forma que os desenvolvedores usam contêineres e componentes funcionais para construir um software baseado em componentes, preparado para evolução não-antecipada, eles podem usar as entidades do paradigma de agentes definidas na AMS para realizar a engenharia de sistemas multiagentes abertos. No entanto, este mapeamento deve ser transparente para o desenvolvedor, ou seja, dado que uma determinada abordagem de composição foi definida para o desenvolvimento, apenas as entidades desta abordagem devem ser consideradas no momento da composição (ALMEIDA, 2005).

Almeida (2005) define a abordagem para composição de software multiagentes em uma visão multi-dimensional em relação aos diversos paradigmas de desenvolvimento de software, como ilustrado na Figura 4. A idéia é preservar as características e facilidades de cada um deles, utilizando agentes e componentes como paradigmas de composição e objetos e aspectos como de programação.

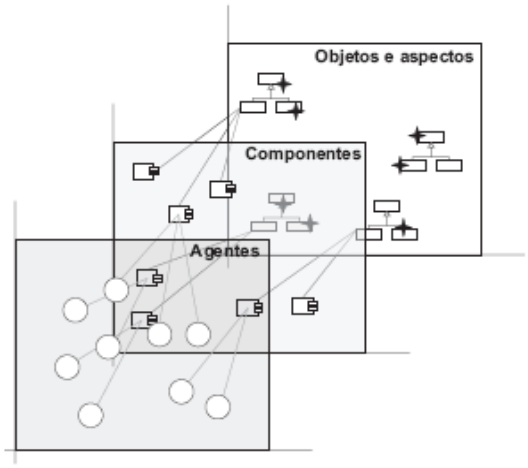

Figura 4 - Visão multi-dimensional de composição (ALMEIDA, 2005)

As entidades definidas na especificação AMS são divididas em três partes: *ambiente*, que define as entidades conceituais relacionados aos ambientes de SMA; *agente*, que define as entidades conceituais relacionadas aos agentes, tais como planos, habilidades, conhecimento, entre outras; e *organização*, que define as entidades relacionadas às interações entre os agentes, tais como leis, papéis, protocolos, entre outras. Como citado anteriormente, este trabalho foca somente nos aspectos de ambientes da especificação AMS, uma vez que o objetivo é conceber e implementar uma infra-estrutura para engenharia de ambientes de SMA.

Conforme discutido no Capítulo 2, existem várias definições de ambientes de SMA, assim como das entidades que o compõem, para os diversos domínios de aplicação. No caso de futebol de robôs, por exemplo, o ambiente deve possuir um mapeamento das posições de cada recurso disponível, ou até mesmo dos agentes que nele estejam imersos (FERBER, 1999). Em outros domínios, porém, a noção de ambiente como um "mapa" pode não fazer sentido, como por exemplo, em sistemas de informação. Neste caso, pode-se utilizar o ambiente como provedor de informações, com acesso a recursos armazenados em repositórios de dados. Vale salientar que neste trabalho o ambiente é definido como uma abstração de primeira ordem, que provê as condições essenciais para os agentes e recursos existirem e que mediam tanto a interação entre os agentes quanto o acesso aos recursos.

Na AMS um ambiente possui um conjunto de recursos, posicionados ou não, que são acessados pelos agentes que compõem o sistema. O acesso aos recursos é controlado por restrições impostas pelo próprio ambiente e pode ser redefinido pelos agentes ou pelo desenvolvedor. Os recursos também podem ser redefinidos, assim como novos podem ser adicionados. Levando em conta estes possíveis cenários, os seguintes requisitos são definidos para flexibilização do projeto de software do ambiente (ALMEIDA, 2005):

1. *Desacoplamento entre agentes e ambiente:* o agente deve ter acesso aos recursos disponíveis no ambiente, de acordo com as restrições de acesso. Porém, o acoplamento Agente-Ambiente e Agente-Recurso deve ser definido de forma a permitir a mudança dos recursos e do ambiente, sem alteração nos agentes. Da mesma forma, mudanças em agentes não devem acarretar mudanças de software nos recursos aos quais tais agentes têm acesso.
2. *Desacoplamento entre recursos:* os recursos disponibilizados no ambiente são definidos de forma genérica, englobando assim qualquer tipo de arquitetura multiagentes, ou pelo menos, grande parte delas. Desta forma, em nível de software, um recurso pode ser tão simples quanto um objeto mantendo um estado, ou tão complexo quanto um gerenciador de banco de dados distribuído. Sendo assim, não há como prever se a aplicação em desenvolvimento irá necessitar de acessos entre recursos. Considerando esta possibilidade, a inserção, remoção ou alteração de um recurso não deve acarretar mudança de software em outros recursos do ambiente.
3. *Definição de restrições através de uma linguagem descritiva:* as restrições de acesso aos recursos devem estar desacopladas do software referente aos recursos ou ambiente. É necessária uma linguagem para descrição das restrições, de forma que tais restrições possam ser alteradas, removidas ou inseridas, sem interferir na execução do sistema.
4. *Desacoplamento entre ambiente e recursos:* em domínios em que o ambiente possua responsabilidades extras em relação aos recursos nele contidos, a unidade de software referente ao ambiente deve poder ser alterada sem acarretar mudanças de software nos recursos.

No nível de ambiente, as entidades de composição definidas na AMS são *Ambiente*, *Recursos* e *Restrições*. Considerando os requisitos enunciados acima, a estrutura de composição do ambiente é apresentada na Figura 5. O *Ambiente* é representado por um contêiner, nos termos da especificação CMS, gerenciando o acesso aos seus recursos e a interação entre eles. Usando como base a especificação CMS, a composição dinâmica de recursos é garantida. Além disso, o ambiente agrega algumas outras características em relação ao contêiner da CMS, tais como: gerência de posicionamento de recursos via mapa/grade e imposição de restrições sobre o acesso aos recursos.

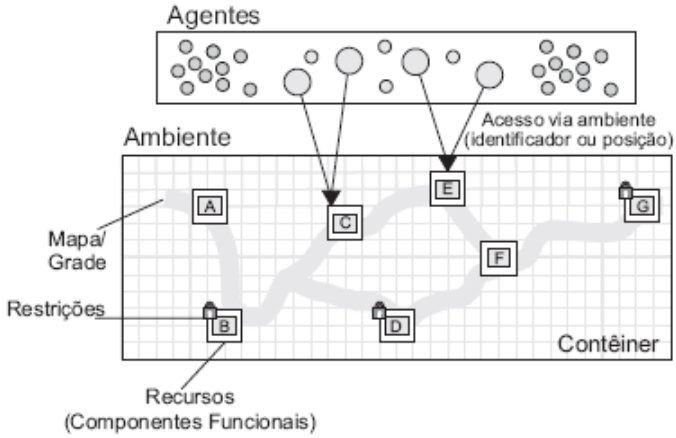

Figura 5 - Estrutura do ambiente na AMS (NUNES et al, 2007)

Os *Recursos* da AMS são representados por componentes funcionais da especificação CMS, cujos serviços básicos são: *leitura*, *escrita*, *remoção* e *criação*. Através da leitura e da escrita é possível obter acesso ao recurso e alterar seu estado. Outros serviços podem ser definidos para os recursos, mas os quatro acima descritos são obrigatórios. Além disso, eventos podem ser disparados e recebidos por recursos, assim como a invocação de serviços entre eles pode ser feita através do ambiente, seguindo a especificação CMS.

Ao contrário da disponibilização de serviços de componentes funcionais, o acesso a recursos pode ser feito via identificador ou através de seu posicionamento. O ponto de acesso aos recursos pelos agentes do sistema é o contêiner do ambiente. Para permitir o posicionamento dos recursos, definem-se os conceitos de grade e mapa. Uma grade é uma matriz de posições para a construção do mapa do ambiente. Um mapa é um subconjunto das posições definidas na matriz, que representam as posições nas quais podem existir recursos (ALMEIDA, 2005). A Figura 7 ilustra um exemplo de definição de mapa para o ambiente ilustrado na Figura 6. Quanto mais fiel à topologia for à descrição do ambiente, maior deverá ser a quantidade de posições na grade.

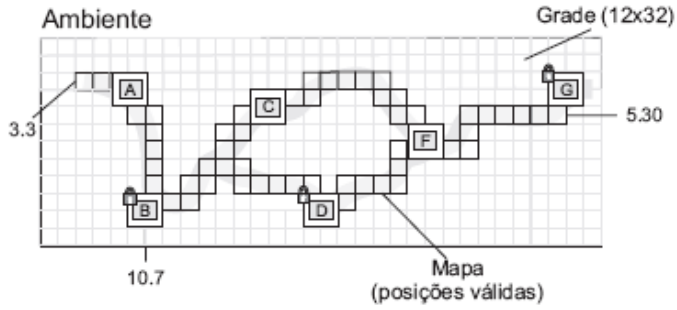

Figura 6 - Definição do mapa do ambiente (ALMEIDA, 2005)

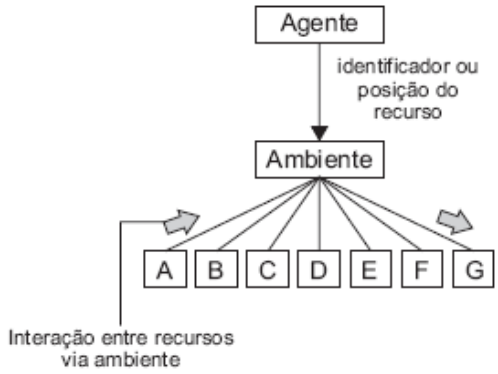

Figura 7 - Estrutura em árvore da composição do ambiente (ALMEIDA, 2005)

Na AMS uma *Restrição* a um recurso é considerada uma adaptação ao provimento de um serviço ou evento por parte do mesmo. Sendo assim, impor restrições de operações sobre um recurso é equivalente a definir um *Adaptador* dos serviços/eventos providos por ele, de acordo com a especificação CMS.

As restrições de acesso básicas, tais como aquelas relacionadas à permissão dos serviços de leitura, escrita, criação e remoção, podem ser gerenciadas pelo ambiente, de uma forma padrão. Uma vez que o adaptador pode ser composto dinamicamente, de acordo com a especificação CMS, as restrições da AMS também podem ser compostas dinamicamente. Além disso, uma restrição pode ser redefinida como um novo adaptador personalizado, caso a restrição às operações básicas não seja suficiente.

De forma resumida, a Tabela 1 apresenta o esquema de mapeamento entre as entidades relacionadas ao nível de ambiente, da especificação de agentes (AMS) à especificação de componentes (CMS).

Entidade de Composição (AMS)	Entidade de Composição (CMS)
Ambiente	Contêiner + mapa
Recurso	Componente Funcional + definição de serviços fixos do recurso
Restrição	Adaptador

Tabela 1 – Mapeamento de entidades do ambiente – AMS e CMS

Capítulo 5. Proposta do Trabalho: Arcabouço de Agentes em Java

O modelo de componentes CMS e o de agentes AMS, apresentados no capítulo anterior, permitem a concepção de sistemas apenas até a fase de projeto, assim como qualquer outro modelo, visto que ele define apenas uma especificação. Para desenvolver e executar um sistema projetado com base em tais modelos torna-se necessário um arcabouço de software que implemente as especificações e conceitos definidos por eles. Segundo Pressman (1999), um arcabouço, ou *framework*, é uma estrutura de suporte definida em que outro projeto de software pode ser organizado e desenvolvido. Arcabouços são projetados com a intenção de facilitar o desenvolvimento de software, habilitando projetistas e programadores a gastarem mais tempo determinando as suas exigências ao invés de se preocuparem com detalhes de baixo nível do sistema. Especificamente em termos de orientação a objeto, *framework* é um conjunto de classes com objetivo de reutilização de um *design*, provendo um guia para uma solução de arquitetura em um domínio específico de software.

No contexto da abordagem baseada em componentes, o *arcabouço* de componentes é a infra-estrutura que possibilita a "montagem" dos componentes para a construção de aplicações, coordenando a interação entre os componentes do sistema (BACHMANN *et al*, 2000). Para garantir que os componentes se comportarão da maneira esperada pelo arcabouço, contratos devem ser definidos. Os contratos representam as interfaces que o componente é obrigado a implementar. Ainda segundo (BACHMANN *et al*, 2000), após o processo de "montagem", os componentes são configurados de acordo com as características específicas da aplicação em questão. Essas informações não são conhecidas enquanto os componentes estão sendo desenvolvidos uma vez que elas são definidas apenas em uma etapa imediatamente anterior à execução da aplicação. A partir do momento que os componentes estão "montados", configurados e integrados ao arcabouço, a aplicação pode então ser executada. Durante a execução da aplicação, os componentes interagem uns com os outros, de quatro formas diferentes: provendo serviços para outros componentes, acessando os serviços destes, gerando e observando eventos.

Mediante a alteração dos requisitos da aplicação, novos componentes também podem ser adicionados, de forma a acrescentar novas funcionalidades ao sistema. Para que essas

alterações nos componentes que constituem a aplicação possam ser realizadas sem interferir na execução do sistema, o arcabouço necessita coordenar a interação entre os componentes de forma a possibilitar a composição dinâmica da aplicação. Ou seja, mesmo durante a execução da aplicação, novos componentes podem ser adicionados, removidos, ou alterados, permitindo que a mesma possa evoluir de acordo com as mudanças de requisitos (BACHMANN *et al*, 2000).

Como descrito acima, é necessário que haja um arcabouço para mediar a interação em aplicações cuja arquitetura é baseada em componentes de software. Geralmente esse arcabouço de componentes é disponibilizado para o desenvolvedor da aplicação através de um conjunto de bibliotecas de classes que são utilizadas para a implementação dos componentes. O arcabouço JCF (*Java Component Framework*) (ALMEIDA *et al*, 2006:2), é a implementação atual na linguagem Java (SUN, 2007) da especificação de componentes CMS do Projeto COMPOR.

No entanto, a especificação de agentes AMS não possuía, até então, uma implementação concreta de seus conceitos. A proposta deste trabalho é o desenvolvimento em Java de um arcabouço para a AMS, denominado JAF (*Java Agent Framework*) (NUNES *et al*, 2007). Como a AMS é baseada na abordagem orientada a componentes da CMS, o JAF é uma extensão do arcabouço JCF.

O arcabouço JAF possibilita a composição de software baseado em multiagentes, inclusive aqueles que lidam com ambientes abertos e dinâmicos. Nas seções a seguir são apresentados os dois arcabouços. Primeiramente, é apresentado o JCF, que suporta composição de software para sistemas baseados em componentes, além de ser o arcabouço no qual o JAF se baseia. Em seguida, o JAF será apresentado.

5.1 Arcabouço de Componentes em Java

Como dito anteriormente, JCF (*Java Component Framework*) é a implementação em Java da CMS. O design do JAF é baseado no padrão de projeto *Composite* (GAMMA *et al*, 1995). A Figura 8 apresenta uma versão simplificada do diagrama de classes do JCF, descrevendo seus principais métodos. A notação utilizada no diagrama é especificada pela linguagem UML (*Unified Modeling Language*) (BOOCH et al, 2000).

As classes *Container* e *FunctionalComponent* são instanciadas em contêineres e componentes funcionais, respectivamente. As instâncias da classe *FunctionalComponent*

representam os componentes funcionais definido no modelo e as instâncias da classe *Container* representam os contêineres.

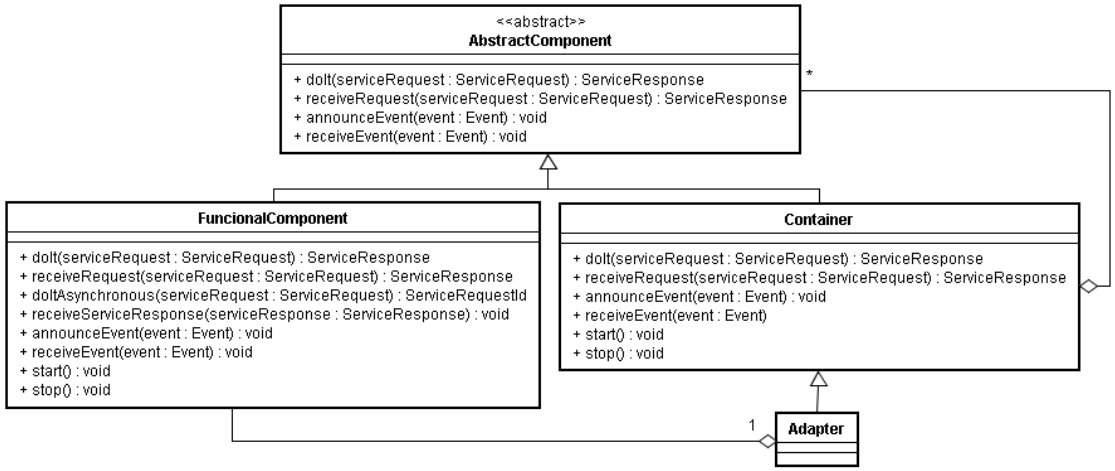

Figura 8 - Diagrama de Classes simplificado do JCF

Essas duas classes herdam a implementação da classe *AbstractComponent*, na qual são implementados os métodos comuns aos componentes funcionais e aos contêineres, tais como os métodos *get* e *set*. A agregação entre *Container* e *AbstractComponent*, permite que *Containers* possam conter *FunctionalComponents* e/ou outros *Containers*, uma vez que os objetos dessas classes também são do tipo *AbstractComponent*. Além disso, a utilização desse padrão (GAMMA *et al*, 1995) facilita a implementação do arcabouço, pois permite aos clientes (*Containers*) tratarem de maneira uniforme objetos individuais (*FunctionalComponents*) e composição de objetos (*Containers*).

Adicionalmente, na classe *AbstractComponent* também são definidas as interfaces dos métodos que devem ser implementados pelos componentes funcionais e pelos contêineres, tais como os métodos de adição e remoção de componentes, serviços e eventos; execução de serviços; anúncio de eventos; e inicialização e finalização dos componentes. Em cada uma das subclasses, *FunctionalComponent* e *Container*, esses métodos são implementados de maneira diferente, de acordo com as especificações definidas no modelo de componentes COMPOR. *Adapter* é uma extensão de *Conteiner*.

Segundo Melo Ferreira (2006), o arcabouço JCF utiliza o mecanismo de *class loaders* (SUN, 2007) disponível na linguagem Java para adicionar classes de componentes que não foram "carregados" inicialmente com a aplicação. Novas classes podem ser acrescentadas ao *classpath* da aplicação durante a sua execução, sem a necessidade de interromper a execução do sistema. Com isso, novos componentes podem ser adicionados, alterados ou removidos da

aplicação. Porém, o mecanismo de carregamento de classes não é capaz de identificar/invocar os serviços/eventos disponibilizados pelos componentes adicionados à aplicação. Para isso, é necessário utilizar um mecanismo de reflexão. A API *Reflection* (SUN, 2007) da linguagem Java permite recuperar e invocar os métodos que representam os serviços e eventos dos componentes. Com isso, é possível invocar serviços e anunciar eventos que não eram "conhecidos" ao iniciar a aplicação.

O modelo de interação baseado em serviço da CMS é implementando no JCF através de invocações iterativas dos métodos *doIt* e *receiveRequest*. Estes métodos são invocados pelos componentes e contêineres da hierarquia até que o componente provedor do serviço seja localizado. A função do método *doIt* é repassar a requisição do serviço, de uma maneira *bottom-up*, até alcançar o contêiner que contém a referência para o provedor. Quando isto ocorre, o método *receiveRequest* é invocado, de uma maneira *top-down*, até alcançar o componente funcional que implementa o serviço. Este processo é ilustrado na Figura 9. A sintaxe para a invocação dos métodos do serviço são *doIt(ServiceRequest)* e *receiveRequest(ServiceRequest)*, onde *ServiceRequest* é um objeto que encapsula o nome do serviço e os parâmetros necessários para executar o serviço. O resultado para a chamada desses métodos é um objeto *ServiceResponse*, que encapsula o resultado da execução do serviço ou uma exceção, caso ela ocorra (ALMEIDA *et al*, 2006:1).

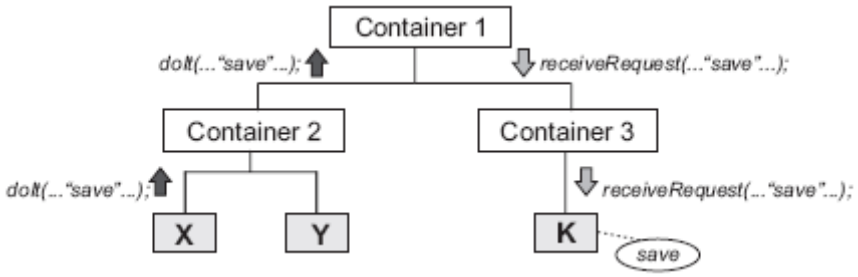

Figura 9 - Execução dos métodos doIt e receiveRequest do modelo de interação baseado em serviços

O JCF também implementa uma versão assíncrona do modelo baseado em serviços. A implementação da interação assíncrona é baseada no padrão de projeto *ActiveObject* (GAMMA *et al*, 1995). Um componente invoca um serviço de modo assíncrono usando o método *doItAsynchronous* e recebe um identificador da requisição (*ServiceRequestId*). Então, uma nova *thread* é iniciada para requisitar o serviço através do método *doIt*. Quando o retorno do método *doIt* ocorre, ele invoca o método *receiveServiceResponse* do componente que requisitou o serviço, repassando a requisição e o identificador do serviço. Baseado no identificador, o componente requisitante identifica para qual invocação a resposta se refere.

Já a implementação do modelo de interação baseado em eventos é baseada nos padrões de projeto *Observer* e *ActiveObject* (GAMMA *et al*, 1995). A funcionalidade é implementada através de invocações assíncronas do método *announceEvent* para anunciar eventos ao contêiner pai (*bottom-up*). Por outro lado, a invocação do método *receiveEvent* notifica a ocorrência dos eventos aos componentes interessados (*top-down*), de forma similar ao modelo baseado em serviços.

5.2 Arcabouço de Agentes em Java

O JAF (*Java Agent Framework*) é uma implementação em linguagem Java da especificação AMS. O JAF concretiza as entidades conceituais presentes na AMS no nível de ambientes de SMA. As principais classes do arcabouço JAF são *Environment*, *MappedEnvironment* e *Resource*. A Figura 10 ilustra uma versão simplificada do diagrama de classes do JAF, descrevendo seus principais métodos e a relação com o arcabouço JCF. Na figura, as classes do JCF estão marcadas com o estereótipo <<jcf>>.

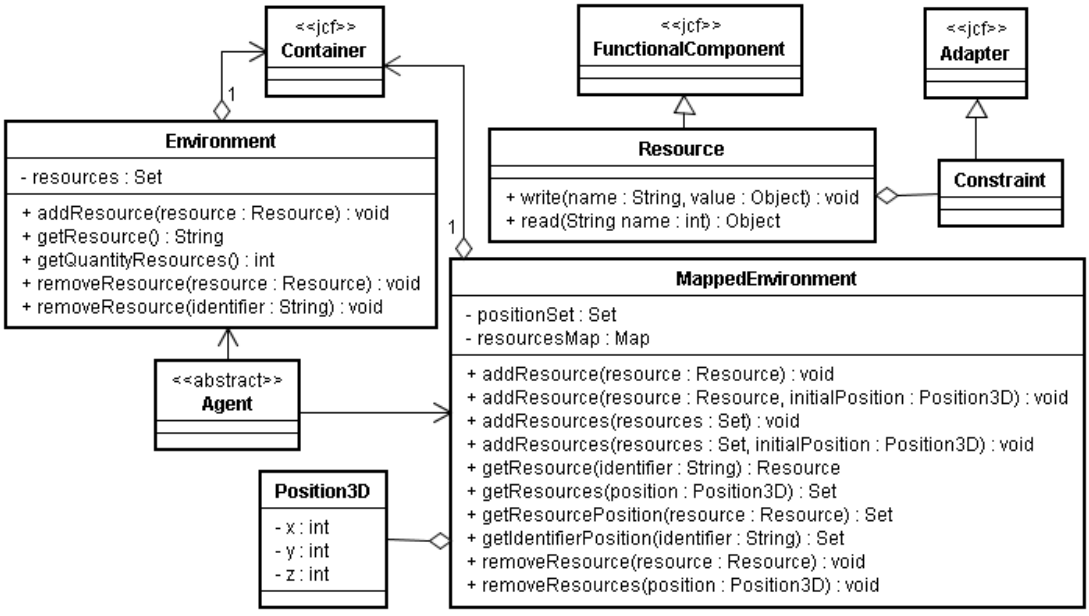

Figura 10 - Diagrama de Classes simplificado do JAF

Instâncias das classes *Environment*, *MappedEnvironment* e *Resource* representam ambientes para SMA e recursos neles contidos. Como pode ser observado na Figura 10, o JAF é totalmente baseado no arcabouço JCF. De acordo com o mapeamento da especificação AMS, as entidades do JAF estendem as principais classes do JCF: *Container* e

FunctionalComponent. Desta forma, as entidades herdam a característica de composição dinâmica deste arcabouço. Além do mais, a infra-estrutura contempla um conjunto de requisitos mínimos definidos na AMS, tais como desacoplamento entre recursos e entre recursos e ambiente, devido, principalmente, as características de composição dinâmica da CMS e, portanto do JCF.

Como não são considerados os aspectos internos de agentes, a classe *Agent* é apenas uma classe comum que tem acesso ao ambiente como cliente. Considera-se um agente como uma entidade caixa preta e não há restrições sobre sua implementação. Há suporte para dois tipos de ambientes: mapeado e não-mapeado. Estes tipos são representados, respectivamente, pelas classes *Environment* e *MappedEnvironment*. As subseções a seguir apresentam as implementações dos dois tipos de ambientes e dos recursos no JAF.

5.2.1 Implementação do Ambiente não-mapeado

Para a classe *Environment*, as posições dos recursos não são importantes. Esta classe pode ser usada para implementar um ambiente representado por uma camada de acesso a banco de dados por exemplo. A classe *Environment* é abstrata e segue o padrão de projeto *Singleton* (GAMMA *et al*, 1995), ou seja, é assegurado uma instância única para esta classe através do uso do método *getInstance*. Ela foi projetada deste modo devido à necessidade de se possuir apenas um ambiente por aplicação.

Ao invés de se utilizar da relação de herança/especialização, a classe *Environment* agrega um *Container*. Na realidade o JAF usa um *ScriptContainer*, um tipo especial de *Container* do JCF que representa a raiz dos componentes da CMS. Em outras palavras, para realizar a composição de uma aplicação utilizando o arcabouço JCF, é necessário inicialmente especificar a estrutura hierárquica da aplicação, com a definição dos contêineres nos quais os componentes funcionais são agrupados. A relação de agregação é utilizada por não haver necessidade de se organizar contêineres hierarquicamente, ou seja, pela inexistência de contêineres filhos, pois o ambiente é representado por apenas 1 (um) contêiner que é composto de vários recursos. Além do mais, é importante ter um maior controle sobre os recursos do ambiente através de seus identificadores, pois a CMS e o JCF não prevêem uso de referências explícitas entre seus componentes. Porém, em nível de ambientes, é imprescindível usar e conhecer os identificadores dos recursos para poder recuperá-los e manipulá-los. Felizmente, a característica de composição dinâmica do JCF é mantida nas interações entre recursos e ambiente. Além de um contêiner, a classe *Environment* possui um

conjunto de recursos como propriedade. Este conjunto armazena os recursos replicados que o contêiner possui, devido ao motivo explicado acima, sendo implementado usando a API *Collections* (SUN, 2007) de Java através do tipo *java.util.Set*.

Os principais serviços providos pela classe *Environment* são recuperação, adição e remoção de recursos. A recuperação de um recurso do ambiente pode ser realizada usando o método *getResource* que o procura no conjunto através do seu identificador e o retorna caso o encontre. Os métodos *addResource* e *removeResource* realizam adição e remoção de recursos, respectivamente, e manipulam tanto a instância de *Container* (usando os métodos *addComponent* e *removeComponent*) como a de *Set* (usando os métodos *add* e *remove*). Ao adicionar um recurso no contêiner, esta mesma operação é realizada novamente sobre o conjunto e um evento é gerado, alertando sobre a inclusão/remoção de um componente do sistema. As operações realizadas pelos métodos supracitados são detalhadas resumidamente no Código 1.

```
public class Environment {
        …
        private ScriptContainer container;
        private Set<Resource> resources;
        ...
        public void addResource(final Resource resource) {
                this.container.addComponent(resource);
                this.resources.add(resource);
                resource.announceAddEvent();
        }
        public void removeResource(final Resource resource)
                        throws ResourceNotFoundException {

                if (!this.resources.remove(resource)) {
                throw new ResourceNotFoundException("Resource does not found");
                }
                this.container.removeComponent(resource);
                resource.announceRemoveEvent();
        }
        ...
}
```

Código 1 – Métodos de adição e remoção de recursos da classe *Environment*

Portanto, *addResource* e *removeResource* usam delegação com o intuito de repassar essas operações para as classes *Container* e *Set*. No caso da remoção, se o recurso não for encontrado, uma exceção *ResourceNotFoundException* será lançada. O método *removeResource* é sobrecarregado para fornecer opções diferentes de remoção.

5.2.2 Implementação do Ambiente Mapeado

Observando-se a Figura 10, a classe *MappedEnvironment* implementa um ambiente mapeado, onde posições de recursos são essenciais. Como *Environment*, a *MappedEnvironment* é *Singleton* (GAMMA *et al*, 1995) e também possui uma agregação de um *Container*. Como explicado na seção anterior, não foi usada herança devido ao fato desta classe possuir instância única e não haver necessidade de contêineres filhos.

A principal diferença para a classe *Environment* é que esta possui uma propriedade de um mapa, que é um conjunto de posições tridimensionais representadas pela classe *Position3D*. A estrutura do mapa é a base para o posicionamento dos recursos no ambiente e define quais posições podem ser ocupadas por eles. A associação de posições para recursos é implementada através do tipo da linguagem Java *java.util.Map* (SUN, 2007). A partir do atributo *resourcesMap* um par chave-valor é usado para construir o mapa. A chave é a posição do recurso no ambiente - instância da classe *Position3D* - e o valor é um conjunto de recursos representado pelo tipo *java.util.Set*, já que se assume que em uma posição pode haver mais de um recurso.

Além das propriedades *container* e *resourcesMap*, a classe *MappedEnvironment* possui mais três atributos: *boundary*, *positionsSet* e *isValid*. O primeiro é do tipo *Position3D* e armazena o limite do ambiente, ou seja, a fronteira das posições possíveis. O segundo representa o conjunto de posições válidas ou inválidas do ambiente dependendo do valor de controle de *isValid*. Através deste atributo *booleano* é possível definir quais são as únicas posições válidas do mapa ou as únicas inválidas. Caso seja necessária a definição de um ambiente que não possa ser adaptado facilmente para as suas posições válidas, pode-se definir um mapa personalizado para o posicionamento dos recursos a partir das inválidas.

A classe *MappedEnvironment* fornece uma diversidade de operações que podem ser utilizadas. Usando o método *setFeatures* é possível definir as características iniciais do ambiente: fronteira e conjunto de posições válidas ou inválidas. Recursos são inseridos no ambiente usando o método *addResource* ou *addResources*. O primeiro é sobrecarregado em duas implementações diferentes: um recurso pode ser adicionado em uma posição específica ou randômica do mapa. Através do método *addResources* é possível adicionar um conjunto de recursos em uma única posição do mapa. Se uma posição inválida é usada, uma exceção *InvalidPositionException* é lançada. Do mesmo modo que na classe *Environment*, ambos os métodos manipulam as propriedades contêiner e mapa da classe, ou seja, ao adicionar ou remover um recurso do contêiner se faz o mesmo no mapa. Instâncias replicadas de *Resource*

são armazenadas com o objetivo de conseguir um maior controle nas posições devido ao JCF não fornecer suporte para esta situação, que se constitui um caso particular dos ambientes mapeados.

Usando o método *getResource* é possível recuperar um recurso do mapa a partir do seu identificador. Já o *getResources* retorna um conjunto de recursos que estão armazenados na posição passada como parâmetro. Em ambos, caso o recurso não seja encontrado, a exceção *ResourceNotFoundException* é lançada. O método *getResourcePosition* retorna as posições ocupadas pelo recurso passado como parâmetro. O método *getIdentifierPosition* também retorna as posições ocupadas pelo recurso, mas a partir de seu identificador, que é passado como parâmetro.

Três métodos privados são implementados nesta classe. O método *isValidPosition* verifica se uma dada posição é válida no mapa do ambiente. O método *exceedsBoundary* verifica se uma dada posição excedeu os limites do ambiente. E o método *sortPosition* escolhe aleatoriamente uma posição válida para ser ocupada por um recurso. Estes métodos são importantes para as operações de adição e remoção de recursos a partir de uma posição no mapa.

Finalmente os métodos *removeResource* e *removeResources* são utilizados para excluir recursos do ambiente, sendo que o segundo remove todos os recursos armazenados em uma dada posição.

Os algoritmos apresentados a seguir são utilizados nas principais operações de adição e exclusão de recursos do mapa, conforme explanado nesta seção.

```
mapaRecursos ← mapa com os recursos do ambiente
recurso ← o recurso a ser adicionado
posicaoInicial ← posição que o recurso será adicionado
contêiner ← o contêiner de recursos
se posição Inicial é válida então
        se posicaoInicial Є mapaDeRecursos então
                se recurso Є mapaDeRecursos então
                        conjuntoRecursos ← conjuntos de recursos da posicaoInicial
                        inclui recurso no conjunto de recursos da posição
                fim se
        senão
                inclui posição e inclui o recurso na posição
        fim senão
        inclui recurso no contêiner
fim se
```

Algoritmo 1 – Adição de um recurso em uma posição do ambiente.

```
identificador ← identificador do recurso a ser recuperado
mapaDeRecursos ← o mapa dos recursos
para cada conjunto de recurso Є mapaDeRecursos faça
        conjuntoRecursos ← o conjunto de recursos de uma posição do mapa
        para cada recurso Є conjuntoRecursos faça
                recurso ← o recurso do conjunto de recursos
                nomeRecurso ← o nome do recurso
                se nomeRecurso = identificador então
                        return recurso
                fim se
        fim para
fim para
```

Algoritmo 2 – Recuperação de um recurso do ambiente.

```
identificador ← identificador do recurso a ser removido
mapaDeRecursos ← o mapa dos recursos
contêiner ← o contêiner de recursos
para cada conjunto de recurso Є mapaDeRecursos faça
        conjuntoRecursos ← o conjunto de recursos de uma posição do mapa
        para cada recurso Є conjuntoRecursos faça
                recurso ← o recurso do conjunto de recursos
                nomeRecurso ← o nome do recurso
                se nomeRecurso = identificador então
                        remove recurso do contêiner
                        remove recurso do mapa de recursos
                fim se
        fim para
fim para
```

Algoritmo 3 – Exclusão de um recurso de uma posição do ambiente.

5.2.3 Implementação do Recurso

A classe *Resource* estende a classe *FunctionalComponent* do JCF para garantir composição recursiva dentro do contêiner do ambiente. Ela possui um atributo identificador e implementa dois métodos principais: *write* e *read*. Estes métodos invocam o método *announceEvent* da superclasse, que implementa a interação baseada em eventos, permitindo que os recursos obtenham e modifiquem propriedades de outros recursos. Em alguns contextos pode ser necessário que os recursos interajam entre si, como por exemplo, em um ambiente composto de sensores em rede. Novos recursos do projeto de software são criados estendendo-se a classe *Resource*.

Os recursos armazenam informações que podem ser acessadas por um agente ou por outros recursos. Para salvar uma informação em um recurso utiliza-se o método *write*. Este método recebe como parâmetro o nome e o valor da propriedade que se deseja salvar. Por outro lado, o método *read* é utilizado para ler uma informação do recurso a partir do nome de sua propriedade. O Código 2 lista os principais trechos da classe *Resource* explicados acima.

```
...
public abstract class Resource extends FunctionalComponent {
    ...
    public final void write(final String propertyName,
                            final Object propertyValue) {
        Object oldPropertyValue = super
                                .getInitializationPropertyValue(propertyName);
        try {
            super.setInitializationPropertyValue(propertyName, propertyValue);
            …
        } catch (ComporInvalidInitializationPropertyException e) {
            super.announceEvent(new EventAnnouncement("write", new Object[] {
                                propertyName, oldPropertyValue, propertyValue }));
        }
    }
    ...
    public final Object read(final String propertyName) {
        Object propertyValue = super
                                .getInitializationPropertyValue(propertyName);
        …
        super.announceEvent(new EventAnnouncement("read", new Object[] {
                                propertyName, propertyValue }));
        return propertyValue;
        }
        ...
}
```

Código 2 – Principais trechos da classe *Resource*

Por sua vez, a classe *Constraint*, que aparece no diagrama de classes da Figura 10, estende a classe *Adapter* do JCF permitindo que as operações *read* e *write* possam ser restringidas. Porém, este mecanismo não foi implementado no trabalho devido às dificuldades encontradas em lidar com o modo pelo qual o *Adapter* foi projetado, pois todos os recursos no JAF possuem a mesma assinatura de métodos, ocasionando um comportamento indevido dos adaptadores no momento da execução.

Capítulo 6. Estudos de Casos

Neste capítulo são apresentados estudos de casos com o objetivo de avaliar a implementação do arcabouço para engenharia de ambientes descrito no capítulo anterior. De acordo com as características definidas em (RUSSEL e NORVIG, 1999) foram escolhidos quatro casos diferentes que abrangem todas as possibilidades de tipos de ambientes para SMA.

Contudo, apenas duas aplicações dentre as quatro propostas foram totalmente implementadas: a *Batalha naval* e o *Jogo Mundo Wumpus*. As demais podem servir como demonstração de possíveis cenários para uso prático do arcabouço. Para cada caso são apresentados: uma visão geral que descreve as características do ambiente, o projeto das classes, as telas de execução da aplicação (para aqueles casos que foram implementados) e a descrição do cenário de evolução.

6.1 Batalha Naval

Este caso é um jogo clássico onde os jogadores definem posições para seus navios no oceano e atacam os navios inimigos especificando as posições de ataque. O objetivo é destruir os navios inimigos. Há uma quantidade mínima de tiros para acertar o adversário. A Figura 11 ilustra um possível cenário para esta aplicação.

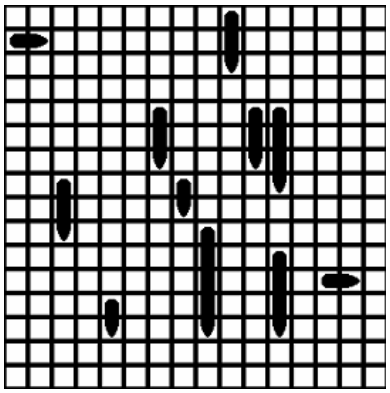

Figura 11 - Cenário de simulação da aplicação Batalha Naval

As características do ambiente descrito são: inacessível, determinístico, episódico, estático e discreto. Este ambiente é inacessível porque o jogador não possui acesso as posições dos recursos do adversário. Ele é determinístico porque o próximo tiro do jogador dependerá da posição do recurso descoberto na rodada atual. Ele é episódico porque existem rodadas de ataque estabelecidas. O ambiente é estático porque os recursos permanecerão em

suas posições todo o tempo e é discreto porque existe um número limitado de ações possíveis no jogo.

Também foram projetadas as classes para esta aplicação estendendo o projeto do JAF no intuito de criar classes específicas. Elas são apresentadas na Figura 12.

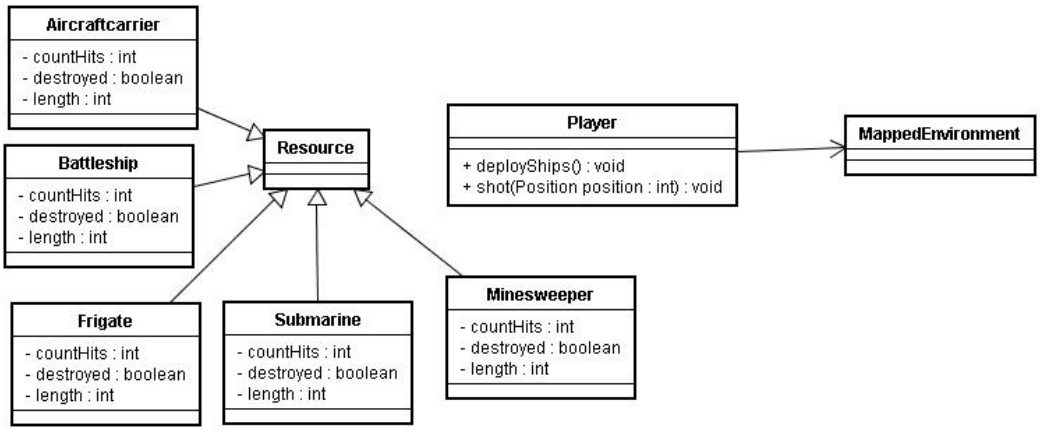

Figura 12 - Diagrama de Classes da aplicação Batalha Naval

A versão pretendida para o jogo da Batalha Naval consiste em: posicionar aleatoriamente onze barcos no mar (ambiente do jogador); aceitar do usuário cliques nas coordenadas do mapa representando seus tiros, sendo indicadas no máximo três posições por vez; verificar se cada disparo acertou um navio ou a água (tiros repetidos são ignorados). A aplicação envia ao usuário mensagens indicando qual o resultado dos tiros (quantos acertaram e quantos falharam). Volta-se a aceitar tiros até toda a frota do adversário ter sido afundada. Ao afundar toda a frota é emitida uma mensagem ao usuário indicando que terminou o jogo e quantos tiros ele disparou.

A seguir são apresentados os detalhes da implementação desta aplicação descrevendo o programa principal, a composição do ambiente e a ilustração das telas em tempo de execução.

Implementação

De acordo com a listagem do Código 3, a linha 6 instancia um ambiente mapeado para ser usado na aplicação. Depois de criados os recursos que farão parte do ambiente (linhas 6 a 12), os mesmos são inseridos usando o método *addResource*. Os tiros do jogador são efetuados usando o método *shot* da mesma classe. Parte da implementação deste método é apresentada no Código 4. Após receber a posição que o tiro pretende atingir, a linha 5 do

Código 4 recupera os recursos que estão nesta posição e marca a variável *shoted* como verdadeira (linha 9) para controlar os recursos que já foram atingidos.

```
1 public class Player {
2         …
3         MappedEnvironment mySea;
4         …
5         public void deployShips(){
6                 mySea = MappedEnvironment.getInstance();
7                 Frigate frigate = new Frigate();
8
9                 Submarine sub = new Submarine();
10                Battleship bship = new Battleship();
11
12                Aircraftcarrier aircraft = new Aircraftcarrier ();
13
14                mySea.addResource(frigate);
15                mySea.addResource(sub);
16                mySea.addResource(bship);
17                mySea.addResource(craft);
18                …
19        }
20 }
```

Código 3 – Composição do ambiente da aplicação Batalha Naval

```
1 public class Player {
2         …
3         public void shot(Position3D position){
4                 mySea = MappedEnvironment.getInstance();
5                 Set<Resource> resources = mySea.getResources(position)
6                 Iterator it = positions.iterator();
7                 while (it.hasNext()) {
8                         Resource resource = (Resource) it.next();
9                         if (resource != null) shoted = true;
10                        …
11                }
12                …
13        }
14 }
```

Código 4 – Método *shot* da classe Player

A interface gráfica com o usuário foi implementada utilizando a API JCF/*Swing* (SUN, 2007). A inicialização do ambiente da aplicação é realizada no método *main* da classe *GUIBattleships*, apresentada no Código 5. A linha 9 deste código apresenta o *frame* da aplicação e monta os componentes gráficos visíveis aos dois jogadores da aplicação. Cada evento do clique do mouse dispara o método *shot* do objeto *player*.

```
1 public class GUIBattleships extends javax.swing.JFrame {
2         …
3         private Player player1;
4         private Player player2;
5         …
6         public static void main(String args[]){
7            java.awt.EventQueue.invokeLater(new Runnable() {
8                   public void run() {
9                               new GUIBattleships().setVisible(true);
10                              }
11        });
12   }
```

Código 5 – Execução da Aplicação Batalha Naval

A Figura 13 apresenta a tela inicial com a disposição gráfica dos navios no mar do jogador da aplicação após a execução do Código 5. Ao clicar nas coordenadas do mar adversário, o jogador dispara um tiro na posição indicada para tentar acertar o navio do inimigo.

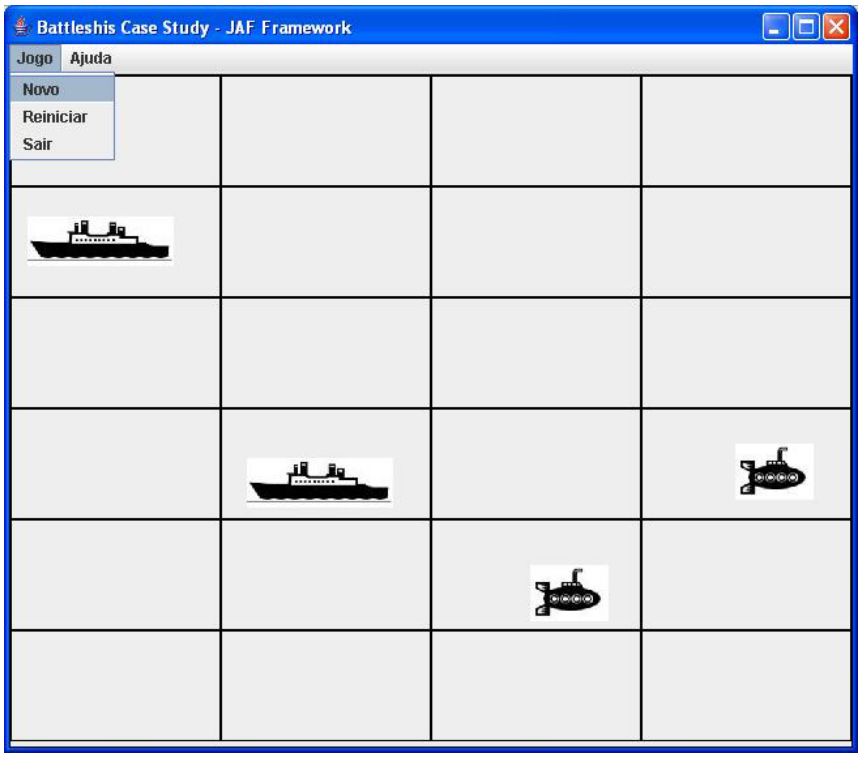

Figura 13 - Tela inicial do Batalha Naval

6.2 Jogo O Mundo de Wumpus

A aplicação em questão se trata da implementação do jogo *Wumpus*, apresentado em (RUSSEL e NORVIG, 1995). O jogo consiste de uma caverna que possui salas conectadas por passagens. Em algum lugar da caverna está o *wumpus*, um monstro que devora qualquer guerreiro que entrar na sala. O *wumpus* pode ser atingido por um guerreiro com o uso de uma flecha. Algumas salas contêm poços sem fundo e um monte de ouro. O agente possui percepções de fedor, brisa, resplendor, impacto ou grito dependendo do objeto a sua frente ser ouro, poço ou o monstro *wumpus*. O objetivo do jogo é matar o *Wumpus*, fugindo das armadilhas encontradas pelo caminho. Manter-se vivo é a principal tarefa para se concluir o objetivo. Deve-se ficar muito atento às indicações de perigo, como por exemplo, o agente é capaz de sentir a brisa que sai dos abismos espalhados pela caverna ou sentir o mal-cheiro exalado pelo terrível *Wumpus*. Uma amostra do mundo de *wumpus* é apresentada na Figura 14.

Figura 14 - Um típico mundo do Jogo Wumpus

O projeto do jogo foi modelado no JAF usando as classes ilustradas na Figura 15. O ambiente é posicionado como uma matriz e está representado pela classe *WumpusEnvironment*, que contém um *MappedEnvironment* do JAF. *Pit, breeze, stench, gold, glitter* e o monstro *wumpus* são representados por classes que estendem *Resource* permitindo composição dinâmica dos recursos encontrados no ambiente. O jogador utiliza a classe *WumpusAgent* para perceber os recursos disponíveis no ambiente e se locomover.

Embora o mundo de *wumpus* seja bastante domesticado para os padrões de jogos atuais, ele constituiu um excelente ambiente para testar as funcionalidades e comportamento do JAF. Neste caso o ambiente é definido como: inacessível, determinístico, episódico, dinâmico e contínuo. Inacessível e determinístico porque o agente não tem acesso aos recursos dispostos na caverna e os seus próximos passos dependem do anterior. É episódico porque existem rodadas dos passos dados pelo agente. É dinâmico porque o agente se

movimenta na caverna e interage com os demais recursos. O ambiente é discreto porque existe um número limitado de ações possíveis no jogo.

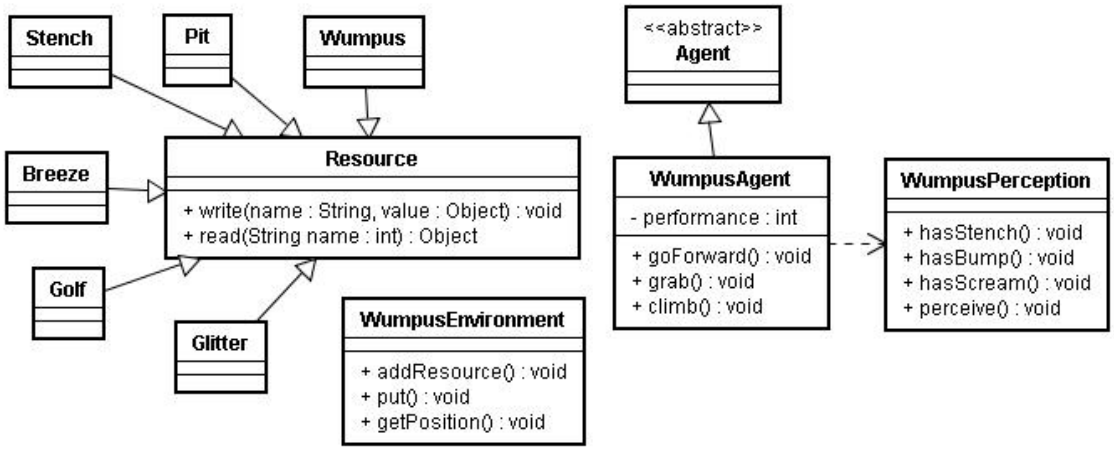

Figura 15 - Diagrama Simplificado do Jogo Wumpus

Para simplificar, o ambiente criado é diferente do proposto por (RUSSEL e NORVIG, 1995) nos seguintes aspectos: o ambiente é não-cíclico, ou seja, não se pode de um lado do ambiente atravessar para o outro lado com apenas uma movimentação; não foi implementado que as flechas percorressem toda a direção na qual elas foram disparadas até se chocarem com uma parede, pois se acredita ser uma característica desnecessária; as percepções do agente se limitam em sentir a brisa dos abismos que estiverem nas casas adjacentes, o mal cheiro do *Wumpus* se ele estiver em uma das casas adjacentes e perceber o brilho das barras de ouro se elas estiverem na mesma casa do agente.

A seguir são apresentados os detalhes da implementação desta aplicação, mais uma vez descreve-se o programa principal, a composição do ambiente e as telas ilustradas em tempo de execução.

Implementação

A classe *WumpusEnvironment* é a responsável por compor o ambiente da aplicação. Entre as linhas 07 e 14 do Código 5 encontram-se as instanciações dos recursos da aplicação um a um. Após sua criação, os recursos são inseridos no ambiente (linhas 16 a 18) usando-se o método *addResource* da classe *MappedEnvironment*. As linhas 20 a 30 recuperam as posições que os recursos foram inseridos para posicionar os demais recursos na posição

correta de acordo com as regras do jogo explanadas anteriormente, para este fim o método *put* é chamado.

```
...
1 public class WumpusEnvironment {
2       ...
3       public void addResources() {
4               Position3D wumpusPosition;
5               Position3D pitPosition;
6               Position3D goldPosition;
7               Wumpus wumpus = new Wumpus();
8               Stench stench = new Stench();
9
10              Pit pit = new Pit();
11              Breeze breeze = new Breeze();
12
13              Gold gold = new Gold();
14              Glitter glitter = new Glitter();
15
16              MappedEnvironment.getInstance().addResource(wumpus);
17              MappedEnvironment.getInstance().addResource(gold);
18              MappedEnvironment.getInstance().addResource(pit);
19              try {
20                      wumpusPosition =
21                              this.getPosition(MappedEnvironment.getInstance()
22                                      .getResourcePosition(wumpus));
23                      this.put(wumpusPosition, stench);
24
25                      pitPosition = this.getPosition(MappedEnvironment.getInstance()
26                                      .getResourcePosition(pit));
27                      this.put(pitPosition, breeze);
28                      goldPosition = this.getPosition(MappedEnvironment.getInstance()
29                                      .getResourcePosition(gold));
30                      this.put(goldPosition, glitter);
31              } catch (ResourceNotFoundException e) {
32                      //e.printStackTrace();
33              }
34      }
35 }
```

Código 6 – Composição do Ambiente no Jogo *Wumpus*

A interface gráfica com o usuário também foi implementada usando a API JCF/*Swing* (SUN, 2007). A inicialização do ambiente da aplicação é realizada no método *main* da classe *GUIWumpus* e é semelhante as linhas apresentadas no Código 5, excetuando-se os objetos *players*.

A tela ilustrada na Figura 16 apresenta a disposição gráfica dos recursos no ambiente após a execução do Código 6. O botão *start* inicia a simulação no mundo *Wumpus.* Através do botão *new* obtém-se um novo mapa aleatório do jogo. Através das setas ou do menu de

movimentações o jogador move o recurso agente em múltiplas direções no mapa no intuito de escapar dos obstáculos proporcionados pelo ambiente.

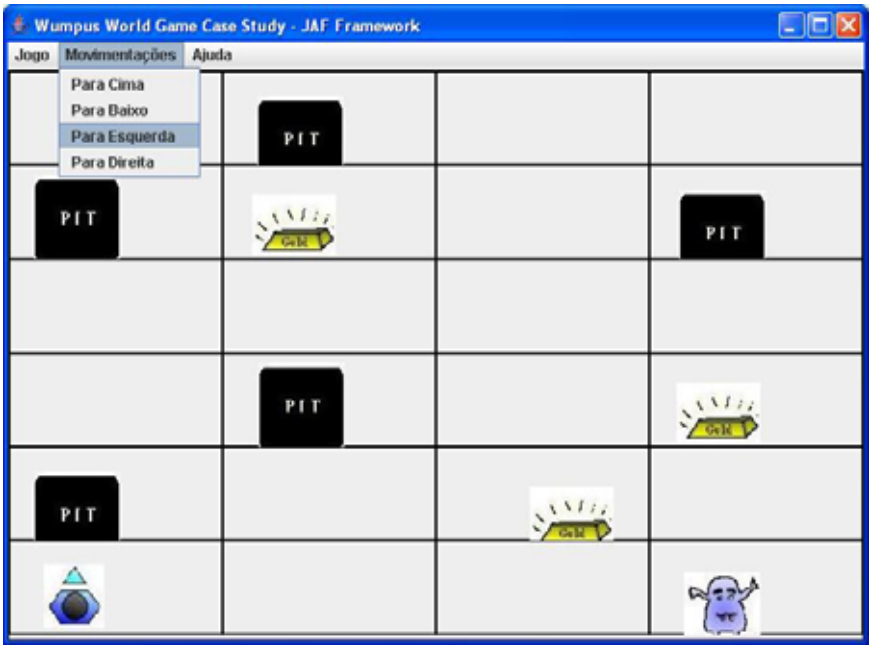

Figura 16 - Tela inicial do Jogo Wumpus

6.3 Simulação de Combate Aéreo

Esta aplicação consiste de uma simulação na qual algumas aeronaves tentam destruir prédios situados em um espaço aéreo inimigo também protegido por aeronaves. A Figura 17 ilustra um possível cenário de simulação.

Figura 17 - Cenário de simulação da aplicação combate aéreo

Este ambiente foi modelado da maneira que se segue. Aeronaves foram consideradas recursos com um plano de vôo pré-definido para atacar ou defender. Prédios também são recursos. As características do ambiente para esta aplicação são: acessível, não-determinístico, não-episódico, dinâmico e contínuo.

O ambiente é caracterizado como acessível porque os recursos podem ser acessados e podem detectar o estado do ambiente devido a movimentação das aeronaves no espaço aéreo. É não-determinístico porque o próximo estado do ambiente não depende da posição corrente e da trajetória dos recursos. É não-episódico porque não há nenhuma quantidade de rodadas estabelecidas de ataques e defesas entre as aeronaves. O ambiente é dinâmico porque o seu estado é representado pelas posições que podem mudar de acordo com a movimentação das aeronaves. O ambiente é contínuo porque não existe um número limitado de movimentos dos recursos.

A Figura 18 ilustra um projeto de classes para esta aplicação, estendendo o arcabouço JAF. São criadas classes específicas da aplicação que podem ser usadas em sua implementação.

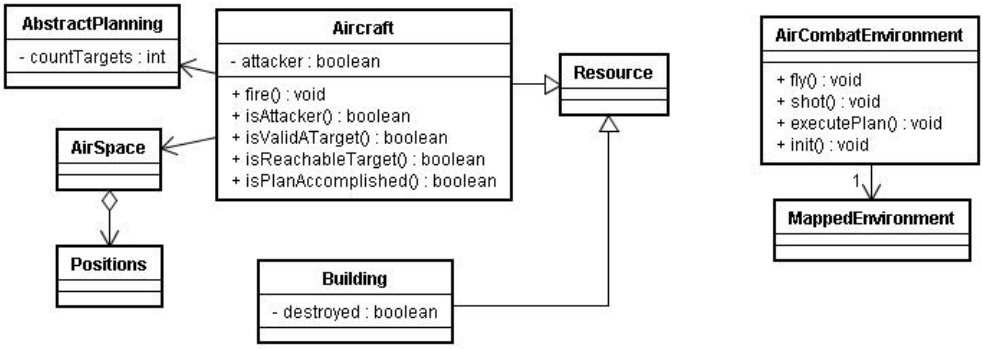

Figura 18 - Diagrama de Classes da aplicação Combate Aéreo

6.4 Enterprise Application Integration

Esta aplicação demonstra o uso do JAF em um caso de integração corporativa de sistemas de informação, também conhecida como *Enterprise Application Integration* (EAI). EAI é considerado um ambiente aberto sem posicionamento de recursos. O objetivo é permitir aos gestores dos sistemas de informação de uma empresa acessar as diversas informações oferecidas pelos dados heterogêneos das aplicações presentes no ambiente, ajudando-os no processo de tomada de decisão da organização.

Um possível cenário para este ambiente é apresentado na Figura 19. *System* são componentes que representam os sistemas integráveis. O componente *ViewHelper* representa o módulo de interface com o usuário que coleta as informações dos sistemas e gera relatórios

e gráficos para os gestores. Por sua vez, o componente *DataAccesLayer* abstrai os detalhes de acesso aos bancos de dados.

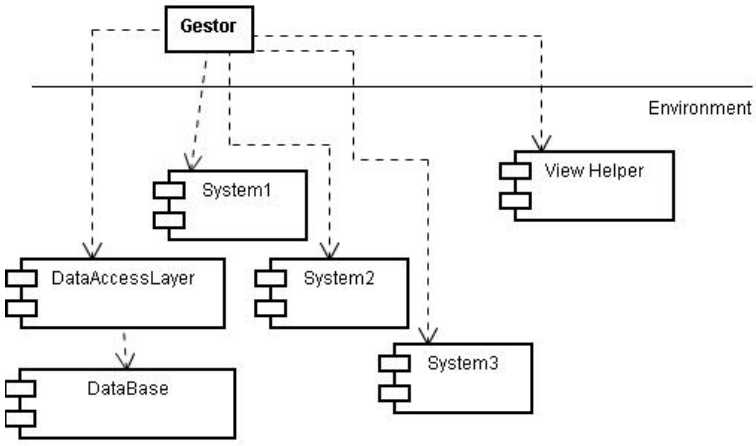

Figura 19 - Cenário da aplicação Integração de Sistemas

Para este estudo de caso a aplicação foi modelada estendendo o JAF a fim de criar as classes específicas, que são representadas na Figura 20. A comunicação com os sistemas existentes é implementada através dos serviços *read*/*write* de instâncias da classe *Provider*. O banco de dados é acessado usando-se a classe *DataBaseLayer*, que é considerada um recurso. Outro tipo de recurso do ambiente é definido pela classe *ViewHelper*. Esta classe é acessada pelos gestores para formatar a informação coletada dos sistemas heterogêneos. A classe *IntegrationEnvironment* representa o ambiente com todos os recursos, neste caso os sistemas integrados. O JAF suporta a evolução deste ambiente tornando possível adicionar novos sistemas ou remover os existentes sem qualquer impacto na aplicação, bem como é possível adicionar novos *ViewHelpers* com implementações diferentes de formatos de visualização de dados.

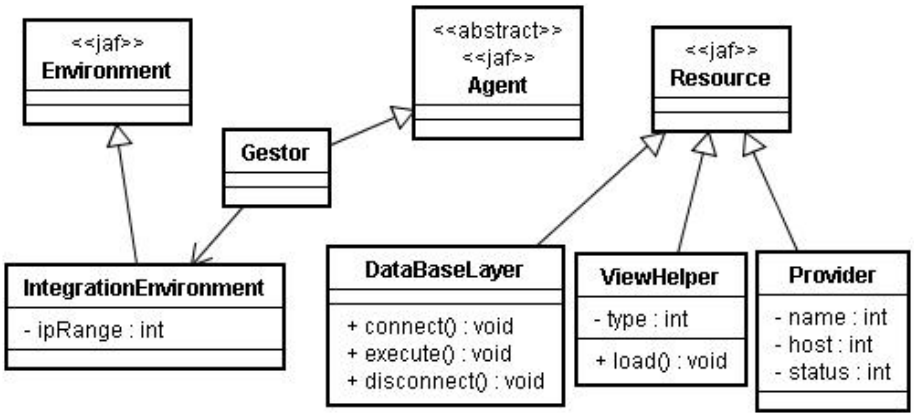

Figura 20 - Diagrama de Classes da Aplicação Integração de Sistemas

Neste caso o ambiente é definido como: acessível, não-determinístico, não-episódico, dinâmico e contínuo. Ele é considerado acessível porque qualquer recurso pode ser lido a qualquer momento pelos agentes. É não-determinístico e não-episódico porque o acesso aos sistemas não dependem de acessos prévios e não há relação entre eles. É dinâmico e contínuo porque o estado do ambiente muda a qualquer momento quando novos sistemas são adicionados ou removidos e, ainda, porque os acessos aos sistemas e o número de potenciais sistemas a serem implantados no ambiente não é limitado.

Capítulo 7. Considerações Finais

O presente trabalho apresentou um estudo minucioso dos papéis do ambiente no projeto e desenvolvimento de SMA, evidenciando que ele é uma parte explícita do sistema e é frequentemente tratado de maneira *ad hoc* pela comunidade. Esta visão *ad hoc* subutiliza as práticas e arquiteturas da engenharia e não explora todo o potencial do ambiente no SMA. Contudo, baseado na definição de (WEYNS *et al*, 2007) este trabalho posicionou o ambiente de SMA como uma entidade de primeira ordem com o intuito de aproveitar o *design* das aplicações que utilizam esta estrutura.

Mostrou-se também que os ambientes de SMA abertos são considerados a classe mais complexa de ambientes. A principal característica é que agentes e recursos entram e saem do sistema dinamicamente, tornando-se necessário flexibilizar o projeto dessas aplicações de modo a permitir cenários de evolução com o mínimo de impacto possível no sistema.

Mediante estas necessidades, este trabalho apresentou um arcabouço, denominado *Java Agent Framework*, para engenharia de SMA abertos, através do qual é possível desenvolver ambientes baseados em Java para sistemas multiagentes, dando suporte a características como dinamicidade, abertura e controle de recursos. O arcabouço implementado seguiu a especificação AMS (*Agent Model Specification*) do Projeto COMPOR. Essa especificação é baseada no modelo de componentes CMS (*Component Model Specification*) – também do mesmo projeto – e tem como objetivo desenvolver software com suporte a evolução não-antecipada, cujo arcabouço é o JCF (*Java Component Framework*). Suportado pela CMS e estendendo o JCF como infra-estrutura de baixo nível, o arcabouço desenvolvido tenta promover uma abordagem genérica e extensível para engenharia de ambientes para diversas modalidades de domínios no contexto de sistemas multiagentes.

A infra-estrutura desenvolvida oferece suporte para a composição dinâmica de aplicações, possibilitando a adição, remoção e troca de componentes (representados por agentes e recursos) em tempo de execução. Por exemplo, um novo recurso que não estava presente no ambiente do SMA pode ser acrescentado à aplicação através da adição de um novo componente e remoção do anterior. Requisitos não previstos durante a concepção da aplicação também podem ser acrescentados com facilidade, visto que a infra-estrutura também possui suporte para evolução de software não antecipada.

A implementação do arcabouço nesse trabalho apresentou as possibilidades de maior empregabilidade da engenharia de ambientes em aplicações multiagentes. A abordagem proposta foi aplicada para desenvolver alguns estudos de casos com a finalidade de identificar potenciais problemas na especificação AMS e, portanto na implementação do arcabouço.

Problemas e algumas limitações foram encontrados no desenvolvimento do arcabouço JAF. Entre eles destacam-se os problemas com identificadores e posicionamento de recursos, pois o arcabouço baseado em componentes que suporta o JAF não permite este mapeamento fixo para garantir a flexibilidade dos componentes. Porém, em ambientes mapeados este aspecto é essencial. Ainda, as restrições de acesso aos recursos que a especificação AMS define não foram implementadas. O motivo foram os problemas encontrados no desenvolvimento dos adaptadores da CMS no JAF devido às operações dos recursos possuírem assinaturas fixas (*read*/*write*).

Em relação aos trabalhos futuros, o arcabouço poderia ser melhorado nos seguintes aspectos:

- Aplicar a abordagem e a infra-estrutura em domínios mais complexos de ambientes para SMA;
- Desenvolver abordagens arquiteturais para o projeto de ambientes, como padrões e referências de arquitetura provendo uma base sólida para o reuso em aplicações de larga escala;
- Explicitar mais ainda as responsabilidades do ambiente e mapear as funcionalidades *core* (comunicação, estrutura, organização) em elementos arquiteturais do projeto do arcabouço;
- Implementar o restante da especificação AMS do Projeto COMPOR nos níveis de agentes e organização, fazendo com que os agentes deixem de ser uma caixa preta na engenharia dos ambientes em aplicações que utilizem o arcabouço apresentado.

Referências Bibliográficas

ALMEIDA, H. O. *COMPOR - Desenvolvimento de Software para Sistemas Multiagentes*. Dissertação de Mestrado, Mestrado em Informática da Universidade Federal de Campina Grande, Campina Grande, PB, Brasil, Março de 2004.

ALMEIDA, H. O.; PERKUSICH, A.; PAES, R. B.; COSTA, E. B. *Composição Dinâmica de Componentes para Aplicações com Mudanças Freqüentes de Requisitos.* In Anais do IV Workshop de Desenvolvimento Baseado em Componentes, volume 4, págs. 9–14, João Pessoa, PB, 2004.

ALMEIDA, H. O. *Composição Dinâmica de Software Baseada em Agentes e Componentes.* Relatório de Projeto de Pesquisa, Departamento de Engenharia Elétrica, Universidade Federal de Campina Grande, Campina Grande, PB, Brasil, 2005.

ALMEIDA, H.; FERREIRA, G.; LOUREIRO, E.; OLIVEIRA, L.; PERKUSICH, A.; PAES, R.; COSTA, E. *A Component Based Dynamic Composition Model to Support Unanticipated Software Evolution*. In: International Conference on Software Engineering (ICSE'06)'. China, 2006.

ALMEIDA, H. O.; PERKUSICH, A.; FERREIRA, G. V. V. M.; FILHO, E. C. L.; COSTA, E. B. *A Component Model to Support Dynamic Unanticipated Software Evolution*. In SEKE'06: Proceedings of International Conference on Software Engineering and Knowledge Engineering, pages 262{267, San Francisco, USA, 2006.

AVANCINI, H.; AMANDI, A. *The Open Agent Architecture: a framework for building distributed software systems*. SADIO Electronic Journal of Informatics and Operations Research **3**(1), pp.1–12, 2000.

BELLIFEMINE, F.; RIMASSA, G.; POGGI, A. *Jade - a fipa-compliant agent framework*. In: Proceedings of the 4th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents, UK, pp. 97–108, 1999.

BACHMANN, F.; BASS, L.; BUHMAN, C.; COMELLA-DORDA, S.; LONG, F.; ROBERT, J.; SEACORD, R. e WALLNAU, K. *Technical Concepts of Component-based Software Engineering*. Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon - Software Engeneering Institute, Maio 2000.

BIANCHINI, C. P.; FONTES, D. S.; PRADO, A. F. *A distributed software agents platform framework.* In: 1st International Workshop on Software Engineering for Large-Scale Multi-Agent Systems - SELMAS'2002 - Orlando/Flórida - USA.

BOOCH, G., RUMBAUGH, J., JACOBSON, I. *UML - Guia do Usuário*. Campus, Rio de Janeiro, RJ, 2000.

BRESCIANI, P.; PERINI, A.; GIORGINI, P.; GIUNCHIGLIA, F.; MYLOPOULOS, J. *Tropos: An agent-oriented software development methodology*. Autonomous Agents and Multi-Agent Systems, 8(3):203{236, 2004.

CHAUHAN, D. *JAFMAS: A Java-based Agent Framework for Multiagent Systems Development and Implementation*. PhD thesis, ECECS Department Thesis, University of Cincinnati, Cincinnati, OH, 1997.

CRNKOVIC, I. *Component-based Software Engineering - New Challenges in Software Development*. In Software Focus, volume 4, pages 127–133. Wiley, Dezembro 2001.

DELOACH, S.A. *Analysis and design using mase and agenttool*. In: Proceedings of 12th Midwest Arti¯cial Intelligence and Cognitive Science Conference, Oxford, Ohio, 2001.

DEMAZEAU, Y. *Multi-Agent Systems Methodology*. In 2nd Franco-Mexican School on Cooperative and Distributed Systems, LAFMI 2003, http://lafmi.lania.mx/escuelas/esd03/ponencias/Demazeau.pdf.

EVANS, R., KEARNEY, P., CAIRE, G., GARIJO, F., GOMEZ SANZ, J., PAVON, J., LEAL, F., CHAINHO, P., MASSONET, P.: *MESSAGE: Methodology for Engineering Systems of Software Agents*. EURESCOM, EDIN 0223-0907 (2001)

FERBER, J. *Multi-Agent System: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley, 1999.

FERREIRA, G.V. ALMEIDA, H.O., PERKUSICH, A., COSTA, E.B. *Especificação e Implementação de Protocolos de Interação entre Agentes para a Plataforma COMPOR*. Infocomp Revista de Ciência da Computação, 3(2):1–7, Novembro 2004.

FIPA. *Foundation for Intelligent Physical Agents*. Disponível em http://www.fipa.org/. Acessado em agosto de 2007.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995.

GRAHAM, J.; DECKER, K. *Towards a distributed, environment-centered agent framework.* In: Proc. of International Workshop on Agent Theories, Architectures, and Languages (ATAL99)', pp. 290–304, 1999.

GOLDSMITH, S.; SPIRES, S.; PHILLIPS, L. *Object frameworks for agent system development*. In: Software Tools for Developing Agents - AAAI Workshop, pp. 107–112, 1998.

HAESEVOETS, R.; VAN EYLEN, B.; WEYNS, D.; HELLEBOOGH, A.; HOLVOET, T.; JOOSEN, W. *Coordinated Monitoring of Traffic Jams with Context-Driven Dynamic Organizations*, International Conference on Engineering Environment-Mediated Multiagent Systems, Dresden 2007.

HEINEMAN, G.T., COUNCILL, W.T. *Component Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, Boston, EUA, 2001.

HOWDEN, W.; RONNQUIST, R.; HODGSON, A.; LUCAS, A. *JACK Intelligent Agents*. The Agent Oriented Software Group (AOS). Disponível em http://www.agent-software.com/shared/home/. Acessado em julho de 2007.

HUBNER, J.M. *Migração de Agentes em Sistemas Multi-agentes Abertos*. Dissertação de Mestrado, Mestrado em Ciência da Computação da Universidade Federal do Rio Grande do Sul, Porto Alegre, Outubro de 1995.

JENNINGS , N. R.. *Agent-Oriented Software Engineering*. In Francisco J. Garijo and Magnus Boman, editors, Proceedings of the 9th EuropeanWorkshop on Modelling Autonomous Agents in a Multi-AgentWorld : Multi-Agent System Engineering (MAAMAW-99), volume 1647, pages 1–7. Springer-Verlag: Heidelberg, Alemanha, 30– 2 1999.

JENNINGS, N. R. *An Agent-based Approach for Building Complex Software Systems*. Commun. ACM 44(4), 35–41, 2001.

KENDALL, E. A.; KRISHNA, P. V. M.; PATHAK, C. V. P.; SURESH, C. B. *A Java Application Framework for Agent based Systems.* ACM Computing Surveys Symposium on Application Frameworks, 2000.

LAU, K-K. *Component-Based Software Development: Case Studies*, volume 1 of Series on Component-Based Software Development. World Scientific Publishing Company, Cingapura, 2004.

McCLURE, C. *Software Reuse: A Standards-Based Guide*. Los Alamitos: IEEE, 2001.

MARTIN, D.; CHEYER, A.; MORAN, D. *The Open Agent Architecture: a framework for building distributed software systems*. Applied Artificial Intelligence **13**(1/2), 91–128, 1999.

MELO FERREIRA, G.V.V. *Infra-estrutura de Software Baseada em Componentes para a Construção de Aplicações para Comunidades Virtuais Móveis*. Dissertação de Mestrado, Mestrado em Informática da Universidade Federal de Campina Grande, Campina Grande, PB, Brasil, Outubro de 2006.

MILI, R., LEASK, G., SHAKYA, U., STEINER, R., OLADIMEJE, E.: *Architectural Design of the DIVAS Environment*. First International Workshop on Environments for Multiagent Systems, New York (2004).

NUNES, C. P. B.; SOUZA JÚNIOR, M. F.; ALMEIDA, H. O.; FERREIRA, G. V. V. M.; PERKUSICH, A.; COSTA, E. B. *Applying a Component-Based Framework to Develop Multi-Agent Environments: Case Studies*. In: 22nd Annual ACM Symposium on Applied Computing (ACM SAC'07). Seoul, 2007.

NWANA, S., NDUMU, D.T., LEE, L.C., COLLIS, J.C.: *Zeus: A Toolkit for Building Distributed Multi-Agent Systems*. 3th International Conference on Autonomous Agents, Seattle, WA, USA (1999).

ODELL, J.; PARUNAK, H. V. D.; FLEISCHER, M.; BRUECKNER, S. *Modeling Agents and Their Environment*. In *AOSE*, Lecture Notes in Computer Science, pages 16{31. Springer, 2002.

PADGHAM, L.; WINIKO, M. *Prometheus: a methodology for developing intelligent agents.* In AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems, pages 37{38, New York, NY, USA, 2002. ACM Press.

PARUNAK, H. V. D. *Go to the Ant: Engineering Principles from Natural Agent Systems*. Annals of Operations Research, 75:69-101, 1997.

PARUNAK, H. V. D.; WEYNS, D. *Environments in Multiagent Systems.* Guest Editorial, Special Issue on Environments in Multiagent Systems, Journal on Autonomous Agents and Multiagent Systems, 14(1), 2007.

PRESSMAN, R.. *Engenharia de Software*. MacronBooks, 1995.

RAO, A; GEORGEFF, M e SONENBERG, E. *Social Plans: A Preliminary Report*. In Decentralized AI 3, 3th European Workshop on Modelling Autonomous Agents in a Multi-Agent World, MAAMAW, Kaiserslautern, Germany, 1992. Elsevier Science B.V.: Amsterdam, Netherland.

RICCI, A.; OMICINI, A.; VIROLI, M.. *CArtAgO: A Framework for Prototyping Artifact-Based Environments in MAS Environments for MultiAgent Systems III*, 3rd International

Workshop (E4MAS 2006), Hakodate, Japan, May 2006. Selected Revised and Invited Papers. LNAI 4389, Springer, February 2007.

RUSSEL, S.; NORVIG, P. *Artificial Intelligence – A Modern Approach*. Prentice-Hall, 1995.

ROGERS, T., ROSS, R., SUBRAHMANIAN, V. *Impact: A system for building agent applications*. Journal of Intelligent Information Systems 13, 1999.

SCHELFTHOUT, K.; HOLVOET, T. *ObjectPlaces: An Environment for Situated Multi-Agent Systems*. In AAMAS '04: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, pages 1500{1501, Washington, DC, USA, 2004. IEEE Computer Society.

SLOMAN, A., POLI, R. *SIM AGENT: A toolkit for exploring agent designs*. Lecture Notes in Computer Science 1037, 1996.

STEINER, R.; LEASK, G.; MILI, R. *An Architecture for MAS Simulation Environments*. In E4MAS '05: Environments for Multi-Agent Systems, pages 50{67, 2005.

SYCARA, K.; PAOLUCCI, M.; VELSEN, M. V.; GIAMPAPA, J. *The RETSINA MAS Infrastructure*. Autonomous Agents and Multi-Agent Systems, 7(1-2):29{48, 2003.

SUN MICROSYSTEMS. *The Java Tutorial.* http://java.sun.com/docs/books/tutorial/, Última atualização agosto de 2007. Acessado em setembro de 2007.

VLISSIDES, J.; COPLIEN, J.; KERTH, N. *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

WEISS, G. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, 1st edition, Julho 2000.

WEYNS, D. e HOLVOET, T. *A formal model for situated multiagent systems*, Fundamenta Informaticae 63 (2-3), pp. 125-158, 2004.

WEYNS, D.; HOLVOET, T. *On environments in multi-agent systems*, AgentLink Newsletter 16, pp. 18-19, December, 2004.

WEYNS, D.; PARUNAK, H. V. D.; MICHEL, F.; HOLVOET, T.; FERBER, J. *Environments for multiagent systems state-of-the-art and research challenges.* In E4MAS '04: The First International Workshop on Environments for Multiagent Systems, volume 3374 of *Lecture Notes in Computer Science*, pages 1{47, New York City, USA, 2004. Springer.

WEYNS, D.; HELLEBOOGH, A.; HOLVOET, T.; *The Packet-World: A Test Bed for Investigating Situated Multiagent Systems.* Software Agent-Based Applications, Platforms and Development Kits, Whitestein Series in Software Agent Technology, 2005.

WEYNS, D.; SCHUMACHER, M.; RICCI, A.; VIROLI, M.; HOLVOET, T. *Environments for Multiagent Systems*. Knowledge Engineering Review, 20 (2) Cambridge University Press, 2005.

WEYNS, D.; PARUNAK, H. V. D.; MICHEL, F. *Environments for Multiagent Systems*. Post-proceedings of the First International Workshop on Environments for Multiagent Systems, Lecture Notes in Computer Science Series, Volume 3374, 2005.

WEYNS, D.; HOLVOET, T. *Multiagent Systems and Software Architecture*. Proceedings of the Special Track on Multiagent Systems and Software Architecture at Net.ObjectDays, 2006.

WEYNS, D.; PARUNAK, H. V. D.; MICHEL, F. *Environments for Multiagent Systems, II*. Post-proceedings of the Second International Workshop on Environments for Multiagent Systems, Lecture Notes in Computer Science Series, Volume 3830, 2006.

WEYNS, D. e HOLVOET, T. A*rchitecture-centric software development of situated multiagent systems*, Engineering Societies in the Agents World VII, Revised Selected and Invited Papers (Dikenelli, O. and Gleizes, M.P. and Ricci, A., eds.), vol 4408, Lecture Notes in Computer Science, pp. 62-85, 2007.

WEYNS, D. e HOLVOET, T. *A framework for situated multiagent systems*, Software Engineering for Large Scale Systems V, Revised Selected and Invited Papers (Choren, R.,

Garcia, A., Giese, H., Leung, H., Lucena, C., and Romanovsky, A., eds.), Lecture Notes in Computer Science, pp. 204-231, 2007

WEYNS, D.; PARUNAK, H. V. D.; MICHEL, F. *Environments for Multiagent Systems, III*. Post-proceedings of the Third International Workshop on Environments for Multiagent Systems, Lecture Notes in Computer Science Series, to appear, Spring 2007.

WEYNS, D.; OMICINI, A.; ODELL, J. *Environment as a First-Class Abstraction in Multiagent Systems*. Special Issue on Environments for Multiagent Systems of Journal on Autonomous Agents and Multiagent Systems, Eds. H. V. D. Parunak and D. Weyns, 14(1), 2007.

WOOLDRIDGE, M. *An Introduction to Multiagent Systems*. Wiley, 2002.

VALCKENAERS, P.; HOLVOET, T. *The environment: An essential abstraction for managing complexity in MAS-based manufacturing control*. In Weyns et al., Springer-Verlag, 2005.

ZUNINO, A.; AMANDI, A. *Brainstorm/J: a Java framework for intelligent agents*. In: Proceedings of the 2*nd* Argentinian Symposium on Artificial Intelligence (ASAI 2000 - 29*th* JAIIO), Tandil, Buenos Aires, Argentina, pp. 43–58, 2000.