1
Introdução

Engenheiros de sistemas multi-agentes (SMAs) móveis devem lidar com o projeto e a implementação de requisitos referentes à mobilidade, além do projeto e implementação de funcionalidades básicas e de outros requisitos, como autonomia, interação e aprendizagem. Conseqüentemente, em meio ao desafio de tratar de questões diretamente relacionadas ao requisito de mobilidade, os engenheiros de SMAs devem considerar a necessidade de produzir sistemas com um nível adequado de *separação de interesses*[1]. Os engenheiros devem garantir que o projeto e a implementação dos *interesses*[2] de mobilidade não estejam entrelaçados com o projeto e a implementação das funcionalidades básicas e dos interesses adicionais em SMAs.

Por outro lado, a importância da modularização de interesses em SMAs torna-se ainda mais evidente, porque a reusabilidade e a manutenibilidade dependem da capacidade de abstrações e mecanismos da engenharia de software em suportar a separação explícita de interesses presentes nos sistemas (Pace et al., 2003). Portanto, as abstrações e mecanismos utilizados devem tornar possível uma maior modularização dos interesses de mobilidade, de modo que a separação de projeto e código obtida limite significativamente o impacto de uma mudança no sistema, além de aumentar suas chances de reuso.

1.1.Problema

Muitas questões importantes referentes à implementação dos interesses de mobilidade devem ser consideradas (Fuggetta et al., 1998; Ubayashi & Tamai, 2001), as quais incluem: (i) a especificação de quais agentes são móveis, (ii) as circunstâncias nas quais os agentes devem se mover, (iii) questões inerentes à

1 Do inglês: *separation of concerns*.
2 Do inglês: *concerns*.

partida de agentes para *hosts* remotos, (iv) ao retorno de agentes para o *host* original, (v) à manutenção do itinerário de agentes, e assim por diante.

Além disso, o código referente às questões de mobilidade, quando implementado por meio das tecnologias atualmente disponíveis, tende a *entrecortar*[3] métodos e classes implementando outras características de SMAs, tais como funcionalidades básicas e colaboração com outros agentes. De fato, os problemas de espalhamento do código de mobilidade em várias classes de um sistema e de entrelaçamento da mobilidade em uma mesma classe com outros interesses do sistema dificultam a produção de SMAs reutilizáveis e manuteníveis. A Figura 1 ilustra os problemas de espalhamento e entrelaçamento da mobilidade com as funcionalidades básicas de SMAs.

Na Figura 1, à esquerda superior, é ilustrado o código de uma classe que corresponde a um agente estacionário de um sistema. À direita, é ilustrada uma classe correspondente ao agente estacionário com código adicional para a implementação dos interesses de mobilidade. O código na parte inferior à esquerda corresponde à manutenção do itinerário do agente móvel definido à direita. A primeira observação importante é que o código da direita é muito maior que o da esquerda superior, embora se trate de dois agentes com as mesmas funcionalidades básicas. Este fato não é surpreendente, uma vez que a implementação dos interesses de mobilidade envolve questões adicionais como definição das circunstâncias nas quais o agente deve se mover, detalhes da partida do agente para *hosts* remotos, detalhes do retorno do agente, etc.

Entretanto, após uma análise mais cuidadosa no código do agente móvel e no código de manutenção do itinerário deste agente, algumas conclusões importantes são obtidas: (i) há perda de encapsulamento das funcionalidades básicas que o agente oferece; (ii) existe um entrelaçamento do código de mobilidade com linhas de código referentes à implementação das funcionalidades básicas do agente; (iii) existe um espalhamento do código de mobilidade por várias classes do sistema, as quais são responsáveis pela implementação dos interesses de mobilidade e possivelmente também apresentando problemas de entrelaçamento. A terceira conclusão pode ser obtida a partir da observação do código referente à manutenção de itinerário e dos vários relacionamentos de

3 Do inglês: *crosscut*.

dependência com outras classes que o agente móvel apresenta no conjunto de seus atributos internos. Tais classes têm a coesão entre os métodos internos diminuída e são altamente acopladas com a plataforma de mobilidade em uso.

```
public class Searcher {

private KnowledgeSource ks;
public SearcherAgent() {
    initKnowledgeSource();
}
public void initKnowledgeSource() {
    ks = KnowledgeSource.getInstance();
    if (ks.disconnected){ks.connect();}
}
public String search(String keyword) {
Enumeration tuples = ks.elements();
boolean foundKeyword = false;
String tuple = new String();
    while ((!foundKeyword) &&
    (tuples.hasMoreElements())) {
        tuple =
        (String)tuples.nextElement();
        int response =
        tuple.indexOf(keyword);
        if (response != -1) {
        foundKeyword = true; }
}
if (foundKeyword) {return tuple;}
else {return "Not successful!"};
}
}
```

```
public class Searcher extends JADEAgent{

private KnowledgeSource ks;
private Itinerary itinerary;
public SearcherAgent() {
    initKnowledgeSource();
    initItinerary();
}
public void initKnowledgeSource() {
    ks = KnowledgeSource.getInstance();
    if (ks.disconnected) {ks.connect();}
}
public String search(String keyword) {
    Enumeration tuples = ks.elements();
    boolean foundKeyword = false;
    String tuple = new String();
    while ((!foundKeyword) &&
    (tuples.hasMoreElements())) {
        tuple =
        (String)tuples.nextElement();
        int response =
        tuple.indexOf(keyword);
      if (response != -1) {
           foundKeyword = true; }
    }
    if (foundKeyword) {return tuple;}
    else { prepareDepartNotification();
           itinerary.goToNext();
           return "Not successful!"
    };
}
public void initItinerary(){
    itinerary = new Itinerary();
    itinerary.init();
}
public void goToNext(){
    itinerary.goToNext();
}
public Message
prepareDepartNotification(){...}
public Message
prepareArrivNotification(){...}
public void listBeliefAgents() {...}
public void beforeMove() {
    prepareDepartNotification();
}
public void afterMove() {
    itinerary.configure();
    clearAgentList();
    listBeliefAgents();
    prepareArrivNotification();
}
}
```

```
public class Itinerary implements
    Serializable{
private JADEAgent agent;
private Vector hosts, visitedHosts,
neighHosts;
private String home, currentHost = null;
public Itinerary(JADEAgent agent) {...}
public void init(){...}
public void configure(){...}
public void addVisited(String host){...}
public void addNeighbor(String host,
neigh) {...}
public Vector getNonVisitedHosts() {...}
public String getNextDestination() {
String nextDestination = this.home;
    Vector neighbors = (Vector)
    neighHosts.get(this.currentAddress);
    ...
    return nextDestination;
}
public void goToNext() {
String next = getNextDestination();
agent.move(next);
}
}
```

LEGENDA: Funcionalidades Básicas Mobilidade

Figura 1. Entrelaçamento de Interesses no Código de SMAs

Os problemas de entrelaçamento e espalhamento de código diminuem o grau de reusabilidade e manutenibilidade dos sistemas (Pace et al., 2003). Isto acontece porque, à medida que a complexidade dos SMAs aumenta, questões referentes à mobilidade não podem ser modularizadas somente a partir da

utilização de abstrações e mecanismos da orientação a objetos (OO). Nesta situação, a implementação dos interesses de mobilidade requer mudanças intrusivas no código das funcionalidades básicas de agentes e a perda de modularidade introduz outros problemas indesejáveis como alto acoplamento e baixa coesão em SMAs. Além disso, estes problemas tornam difícil a transformação não intrusiva de agentes estacionários em agentes móveis.

Apesar das evidências empíricas dos problemas de entrelaçamento e espalhamento na implementação de questões específicas de mobilidade, os desenvolvedores de SMAs têm se apoiado vastamente na utilização de interfaces de programação de aplicações (APIs) orientadas a objetos de plataformas de agentes móveis e na linguagem de programação Java. O resultado é a produção de SMAs que são difíceis de entender, manter e reutilizar.

Este trabalho se propõe a responder a seguinte questão: como implementar SMAs sem que as funcionalidades básicas dos agentes estejam entrelaçadas com a implementação de questões específicas dos interesses de mobilidade? Os requisitos abaixo devem ser considerados:

- O projeto de SMAs deve suportar a introdução transparente de mobilidade em agentes inicialmente estacionários.
- O projeto e a implementação de funcionalidades básicas de SMAs não devem sofrer maiores impactos com as sucessivas manutenções no comportamento de mobilidade de um ou mais agentes.
- O projeto e a implementação de funcionalidades básicas de SMAs não devem sofrer maiores impactos com possíveis trocas de plataformas de mobilidade e mudanças nas interfaces dos componentes de mobilidade.

1.2. Limitações dos Trabalhos Relacionados

As questões referentes à mobilidade têm sido estudadas sob diferentes enfoques, os quais incluem usualmente padrões de projeto (Aridor & Lange, 1998) e plataformas de agentes móveis (Kiniry & Zimmerman, 1997; Lange, 1998; Bellifemine et al., 1999). Entretanto, tais abordagens não possuem como um de seus objetivos centrais a separação explícita do código de mobilidade de outros interesses em SMAs. Em outras palavras, apesar da existência de

plataformas de agentes móveis com diferentes enfoques, estas somente permitem a introdução de mobilidade em SMAs a partir de diretivas inerentemente intrusivas, as quais levam aos problemas de espalhamento e entrelaçamento.

Mais especificamente, a fim de adicionar mobilidade aos agentes, os desenvolvedores geralmente devem modificar o projeto de sistemas:

(1) declarando que os agentes dos sistemas estendem classes de agentes pertencentes às APIs das plataformas de mobilidade;
(2) implementando métodos abstratos declarados em classes das plataformas de mobilidade;
(3) declarando e, possivelmente, definindo a implementação de interfaces pertencentes às APIs das plataformas de mobilidade;
(4) invocando explicitamente métodos das APIs em classes dos sistemas que implementam outros interesses de agentes que não a mobilidade.

Todos as restrições acima diminuem a reusabilidade e a manutenibilidade do sistema, uma vez que a adição ou remoção de código referente à mobilidade necessariamente ocasiona mudanças intrusivas em classes que não possuem como principal finalidade a implementação da característica de mobilidade dos agentes.

É importante notar que não é possível encontrar uma solução com maior grau de modularidade que resolva completamente este problema, mesmo a partir de um processo de refatoração dos sistemas, ou através da utilização de outra plataforma de mobilidade. Isto ocorre porque a mobilidade é um *interesse transversal*[4], isto é, um interesse que naturalmente afeta classes e/ou métodos que modularizam outros interesses nos sistemas. Os interesses transversais, como a mobilidade, não podem ser especificados somente a partir de abstrações e mecanismos de decomposição da orientação a objetos (Ubayashi & Tamai, 2001; Garcia et al., 2004).

De acordo com nosso conhecimento, somente um trabalho na literatura de agentes móveis procurou demonstrar uma abordagem alternativa através da qual se torna possível dar suporte à introdução transparente de mobilidade em aplicações. Trata-se da abordagem *RoleEP* de Ubayashi e Tamai implementada pelo *framework Epsilon/J* dos mesmos autores (Ubayashi & Tamai, 2001). Para o

4 Do inglês: *crosscutting concern*.

uso desta abordagem, os desenvolvedores de SMAs devem necessariamente estender várias interfaces e classes abstratas do *framework Epsilon/J*, o que diminui a modularização dos interesses de mobilidade. Outra limitação importante é que *Epsilon/J* é dependente da plataforma *Aglets*. A abordagem *RoleEP* e o *framework Epsilon/J* são descritos em detalhes no Capítulo 3 deste trabalho.

1.3. Solução Proposta

Existem várias abordagens para a separação avançada de interesses, incluindo programação orientada a sujeitos (Harrison & Ossher, 1993), filtros de composição (Aksit et al., 1993; Bergmans & Aksit, 2001), programação adaptativa (Lieberherr, 1996; Lieberherr et al., 2001) e separação multi-dimensional de interesses (Tarr & Ossher, 2000). Tais abordagens contribuem para melhorar o desenvolvimento de software orientado a objetos, uma vez que proporcionam a separação de interesses em outras dimensões, além de classes e objetos. Elas introduzem novas abstrações de modularização e mecanismos de composição para melhorar a separação de interesses transversais.

A orientação a aspectos (OA) (Kiczales et al., 1997) é uma das abordagens para a separação avançada de interesses. *Aspectos*[5] são usados como abstrações capazes de capturar de forma modular os interesses transversais nos sistemas. *AspectJ* (Kiczales et al., 2001) é uma extensão orientada a aspectos da linguagem de programação Java. *AspectJ* é considerada a linguagem orientada a aspectos mais madura, uma vez que possui uma grande comunidade de usuários, evoluiu muito nos últimos anos e já foi aplicada em projetos da indústria (AspectJ Team). Muitos estudos encontrados na literatura exploram a programação orientada a aspectos (POA) em *AspectJ* (Lippert & Lopes, 2000; Soares et al., 2002; Rashid & Guitchyan, 2003; Garcia et al., 2004). Entretanto, nenhum destes experimentos investigou o uso de POA para o tratamento modular de mobilidade de código.

5 Do inglês: *aspects*.

A POA está evoluindo sistematicamente da etapa de implementação em direção às fases de arquitetura e projeto de sistemas, a fim de fornecer um caminho completo de desenvolvimento ao longo do ciclo de vida de software. O Desenvolvimento de Software Orientado a Aspectos (DSOA) é uma área de pesquisa emergente cujo objetivo é promover a separação avançada de interesses em todas as etapas do desenvolvimento de software.

Portanto, uma possível solução para o problema de modularização de mobilidade em SMAs, investigada neste trabalho, envolve o DSOA. Propõe-se: (1) uma arquitetura de software orientada a aspectos, denominada ArchM[6], que define componentes de software e suas respectivas interfaces, com o objetivo de possibilitar a separação explícita de interesses de mobilidade em sistemas multi-agentes; (2) um *framework* orientado a aspectos, denominado AspectM[7], que implementa a arquitetura ArchM. Tanto na arquitetura ArchM quanto no *framework* AspectM, aspectos são usados como abstrações que modularizam a mobilidade de agentes nas aplicações.

ArchM modulariza os interesses de mobilidade no nível de arquitetura de um software. Em outras palavras, no nível arquitetural, os componentes de mobilidade da arquitetura separam os interesses de mobilidade dos componentes correspondentes às funcionalidades básicas e outros interesses dos sistemas. Cada componente arquitetural de ArchM corresponde a um interesse específico de SMAs e obedece a um conjunto de responsabilidades. As responsabilidades impostas pela arquitetura ArchM podem ser detectadas a partir das interfaces especificadas, as quais tornam acessíveis os serviços implementados pelo componente. Por essa razão, outro beneficio de ArchM é guiar os projetistas no desenvolvimento de SMAs móveis, uma vez que esta arquitetura oferece diretivas quanto à especificação de interfaces, componentes e a interação entre estes. Portanto, a utilização de ArchM pode impactar o grau de reusabilidade e manutenibilidade de SMAs móveis.

6 Do inglês: *Architecting Mobility*.
7 Do inglês: *Aspectizing Mobility*.

O *framework* AspectM modulariza os interesses de mobilidade nos níveis de projeto detalhado e implementação de SMAs. Mais especificamente, o *framework* AspectM dá suporte a: (1) uma clara separação dos interesses de mobilidade em relação às funcionalidades básicas e outros interesses dos agentes, (2) uma introdução transparente do código de mobilidade em agentes estacionários, e (3) uma integração flexível dos SMAs com várias plataformas de mobilidade. O *framework* AspectM foi identificado e abstraído da instanciação de diferentes aplicações de agentes móveis (Garcia, 2004; Barbosa & Goldman, 2005) e a partir do estudo de algumas das principais plataformas de mobilidade existentes (Lange, 1998; Bellifemine et al., 1999). AspectM foi implementado em *AspectJ* e pode ser aplicado para a instanciação de várias aplicações baseadas em agentes.

1.4. Objetivos

Este trabalho ilustra por meio de estudos de caso o uso de um *framework* orientado a aspectos para a implementação dos interesses de mobilidade em SMAs. São objetivos específicos:

1. Levantamento de problemas inerentes à utilização de abstrações e mecanismos OO na implementação dos requisitos de mobilidade de sistemas multi-agentes.
2. *Survey* e análise das soluções existentes para a modularização de mobilidade em SMAs: *frameworks* OO associados com plataformas de agentes móveis (Lange, 1998; Bellifemine et al., 1999), padrões de projeto (Aridor & Lange, 1998) e a abordagem alternativa *RoleEP* (Ubayashi & Tamai, 2001).
3. Proposta de uma arquitetura orientada a aspectos para a modularização dos interesses de mobilidade em SMAs.
4. Proposta e implementação de um *framework* orientado a aspectos seguindo as diretivas da arquitetura acima.
5. A utilização do *framework* de mobilidade em estudos de caso. Atualmente o uso do *framework* abrange as seguintes aplicações: (1) *Expert Committee*, sistema para o gerenciamento de submissões de artigos para conferências, desenvolvido no LES/PUC-Rio; (2)

MobiGrid, que consiste em um *framework* baseado em agentes móveis utilizados em um ambiente de grade computacional, denominado *InteGrade*, desenvolvido no IME-USP.

1.5. Organização do Texto

Este trabalho está organizado como a seguir. No Capítulo 2, são detalhadas as características e funcionalidades básicas de SMAs móveis, sendo que também é apresentado um *survey* de algumas das principais plataformas de mobilidade usadas por estes sistemas. No Capítulo 3, são apresentadas em detalhes todas as soluções propostas na literatura para o problema de separação da mobilidade, analisando vantagens e desvantagens de cada uma. No Capítulo 4, é apresentada uma revisão da literatura sobre DSOA, novo paradigma de desenvolvimento de software utilizado na solução proposta por este trabalho para o problema de separação dos interesses de mobilidade. No Capítulo 5, é apresentada a proposta de uma arquitetura orientada a aspectos como solução para o problema de modularização dos interesses de mobilidade em SMAs. No Capítulo 6, são apresentadas a proposta e a implementação de um *framework* orientado a aspectos seguindo as diretivas da arquitetura descrita no Capítulo 5. A utilização do *framework* de mobilidade em estudos de caso é detalhada no Capítulo 7. Finalmente, no Capítulo 8 são apresentadas as conclusões do trabalho, acompanhadas de uma análise crítica dos resultados encontrados, além de propostas para melhorias e trabalhos futuros.