7
Estudos de Caso

Este capítulo apresenta os estudos de caso utilizados para avaliação do *framework* AspectM. Atualmente a instanciação do *framework* abrange as seguintes aplicações: (1) *Expert Committee* (Garcia, 2004), sistema para o gerenciamento de submissões de artigos para conferências, desenvolvido no LES/PUC-Rio; (2) *MobiGrid* (Barbosa & Goldman, 2005), que consiste em um *framework* baseado em agentes móveis utilizados em um ambiente de grade computacional, denominado *InteGrade*, desenvolvido no IME-USP.

7.1.
Expert Committee

Esta seção detalha a descrição do sistema EC (Seção 3.1), nosso primeiro estudo de caso no processo de abstração do *framework* AspectM. Os agentes de usuário do sistema EC precisam se movimentar em algumas circunstâncias, inclusive quando estão desempenhando papéis específicos (Seção 3.1).

Por exemplo, quando um agente de usuário está desempenhando o papel de *chair*, é necessário consultar dados sobre revisores a fim de otimizar a distribuição de propostas de revisão de acordo os interesses de pesquisa de cada revisor. Se a informação relativa a um ou mais dos revisores não está disponível no banco local, o papel *chair* deve se movimentar para ambientes remotos a fim de encontrar a informação sobre os interesses de pesquisa do revisor. O agente desempenhando o papel *chair* então se movimenta pelos *hosts* da rede delegando a tarefa de busca aos agentes de informação espalhados pela Internet.

A Figura 37 apresenta o diagrama de classes do sistema EC. As classes representam: (i) tipos de agentes (`ResearcherAgent`, `InformationAgent`), (ii) papéis (`Reviewer`, `Chair`) e (iii) planos (`CVUpdatePlan`, `DistributionPlan`, `InformationGatheringPlan`). As classes que implementam os interesses de mobilidade do tipo de agente `ResearcherAgent` e do papel `Chair` utilizam os serviços de mobilidade providos pela plataforma JADE (Seção 2.6).

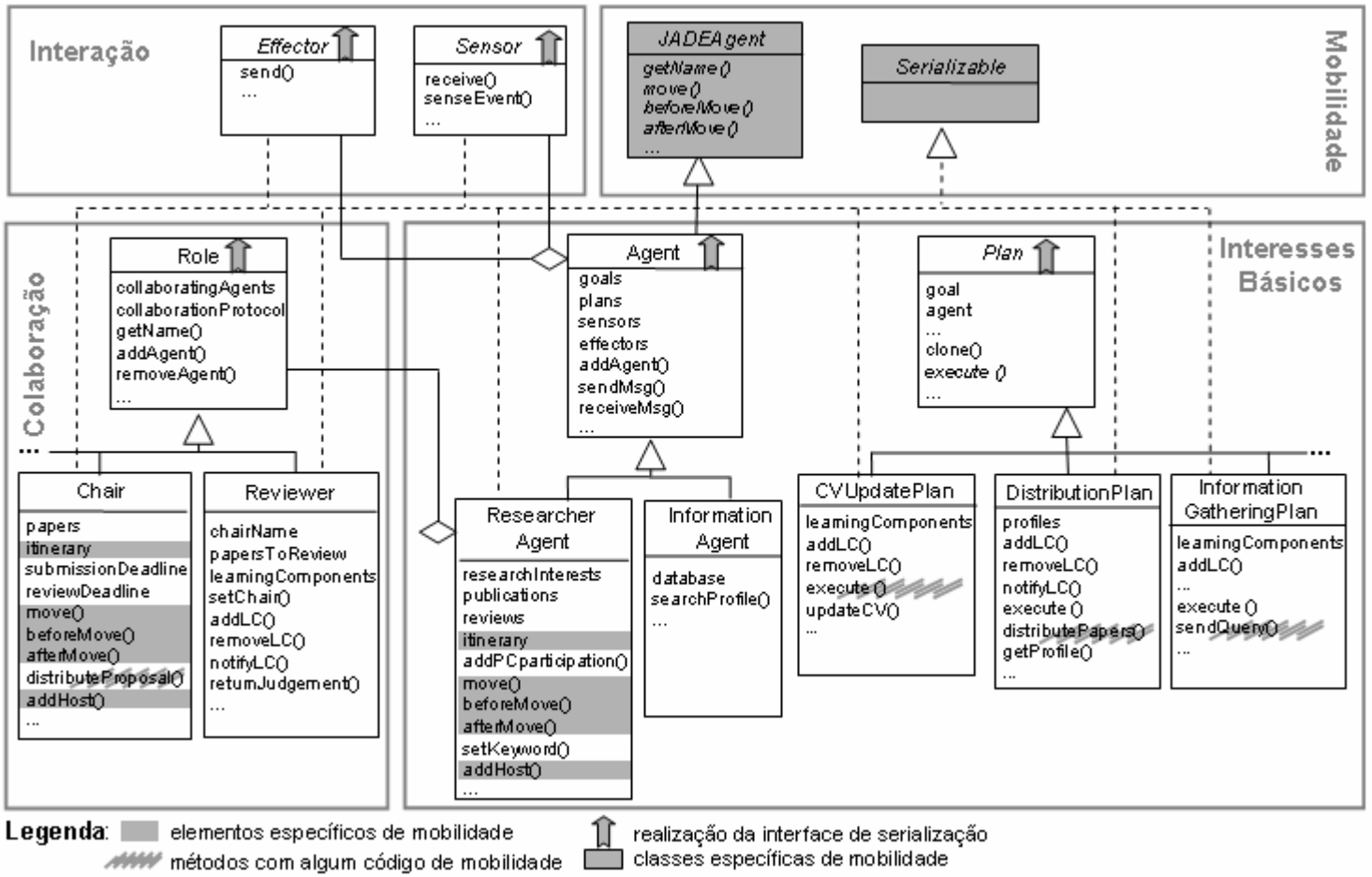

Figura 37. Diagrama de Classes do Sistema Expert Committee

Na Figura 37, os interesses de mobilidade espalhados e entrelaçados no projeto de EC são entrecortados por elementos gráficos em cor cinza (Seção 3.2). Entretanto, no sistema EC, é fundamental que os interesses de mobilidade e os outros interesses do sistema, como as funcionalidades básicas e atividades de colaboração, sejam especificados tão separados quanto possível. Particularmente, para a especificação do papel de *chair*, é necessário definir o protocolo de mobilidade da classe `Chair` de modo a: (i) introduzir de forma transparente as funções de mobilidade necessárias ao papel e (ii) flexibilizar o uso de plataformas que provêem os serviços de mobilidade.

Portanto, podemos utilizar o AspectM (Capítulo 6) para introduzir o comportamento de mobilidade no papel *chair*, tendo em vista o alcance dos objetivos de reusabilidade e manutenibilidade do sistema. As próximas subseções descrevem este processo de introdução de mobilidade no EC.

7.1.1. Descrição do Papel Chair

No domínio de SMAs, um agente pode desempenhar diferentes papéis em uma aplicação (Seção 2.1). Os papéis especificam a maneira como os agentes interagem uns com os outros em relacionamentos de colaboração. Cada papel envolve a definição de um conhecimento extrínseco (Seção 2.1), com crenças, objetivos e planos próprios do papel. Entretanto, o papel também pode manipular o conhecimento intrínseco do agente (Seção 2.1).

No sistema EC, o papel *chair* possui objetivos específicos, incluindo o objetivo de distribuir artigos submetidos a uma conferência para revisão. Para atingir seus objetivos, o papel *chair* pode definir múltiplos planos. Para a execução de seus planos, o papel *chair* possui crenças que armazenam informações sobre a conferência, incluindo datas limites para submissão e revisão de artigos, a lista de revisores e a lista de artigos submetidos. O papel revisor e os outros papéis também possuem crenças, objetivos e planos específicos.

Na Figura 37, subclasses de `Role` são usadas para modularizar múltiplos papéis do tipo de agente `ResearcherAgent`. Neste caso, uma subclasse de `Role` é utilizada para especificar o conhecimento extrínseco correspondente a um papel do agente. Mais especificamente, o conhecimento intrínseco é definido na classe

que representa o tipo do agente, enquanto o conhecimento extrínseco de um papel é representado através de uma subclasse de `Role`.

Atributos e métodos de uma subclasse de `Role` correspondem, respectivamente, às crenças e ao comportamento do papel, e podem se referir a objetivos e planos do próprio papel. Os planos definidos para um papel podem manipular os atributos (crenças) e invocar métodos (comportamento) pertencentes ao agente definidos na classe do papel. No *Expert Committee*, o papel `Chair`, por exemplo, define atributos e métodos adicionais para um agente `ResearcherAgent`, a fim de que este possa assumir o papel de *chair* em um relacionamento de colaboração.

A Figura 38 ilustra a definição do papel `Chair`. São apresentados apenas os atributos e métodos referentes ao comportamento próprio do papel. Os métodos que definem a maneira como o papel `Chair` se relaciona com `ResearcherAgent` não são ilustrados, uma vez que não são importantes para o escopo deste trabalho.


```
1: public class Chair implements Serializable {
2: private ResearcherAgent agent;
3: private DistributionPlan distributionPlan;
4: private List papersList;
5: private List reviewersList;
6: private GregorianCalendar submissionDeadline;
7: private GregorianCalendar reviewDeadline;
8: private Itinerary itinerary;
9: ...
10: public Chair(ResearcherAgent agent){
11:    this.agent = agent;
12:    distributionPlan = new DistributionPlan ();
13:    papersList = new Hashtable();
14:    submissionDeadline = new Calendar();
15:    ...
16: }
17: public DistributionPlan getDistributionPlan(){
18:    return this.distributionPlan;
19: }
20: ...
21: public void addHost(String host){...}
22: }
```

Figura 38. Implementação do Papel Chair no Expert Committee

Na Figura 38, exemplos de atributos definidos em um papel `Chair`: a instância de `ResearcherAgent` associada ao papel `Chair` (linha 2), o plano de distribuição de artigos (linha 3), a lista de artigos submetidos (linha 4), a lista de revisores (linha 5), as datas limites para submissão (linha 6) e revisão (linha 7) de artigos, e assim por diante. A classe `Chair` também define para `ResearcherAgent` os métodos para manipulação dos atributos anteriores (linhas de 17 a 20). Também são ilustrados atributos e métodos referentes ao comportamento de mobilidade do papel `Chair`. Por exemplo, o atributo que define o itinerário do papel (linha 8) e métodos de manipulação do itinerário (linha 21).

A partir da implementação de `Chair` na Figura 38, é razoável concluir que a classe `ResearcherAgent` não possui atributos e métodos correspondentes a crenças e planos pertencentes ao conhecimento extrínseco do papel *chair*. Isto ocorre porque a classe `Chair` é desenvolvida com o objetivo de modularizar o papel *chair* de um agente `ResearcherAgent` no sistema EC. Conseqüentemente, é razoável também concluir que o comportamento de mobilidade de `Chair` é implementado de forma modular. Entretanto, uma vez que no projeto do EC não só as classes de papéis, mas também tipos de agentes ainda estendem classes e interfaces de plataformas de mobilidade, permanecem as restrições arquiteturais impostas ao projeto de SMAs móveis (Seções 3.1 e 3.2). Estas restrições são ilustradas no diagrama de classes da Figura 37. Os interesses de mobilidade estão espalhados e entrelaçados em meio a funcionalidades básicas e interesses de colaboração do sistema EC.

Uma nova definição para o comportamento de mobilidade de `Chair` pode ser obtida utilizando o AspectM (Capítulo 6). O projeto de mobilidade para o sistema EC usando AspectM é apresentado na próxima seção.

7.1.2. Mobilidade do Papel Chair

Nesta seção, supomos que o pacote de classes para uso dos serviços de JADE e *Aglets* já estão disponíveis. Tratamos da instanciação do pacote para uso flexível de plataformas na Seção 6.3. A Figura 39 apresenta a instanciação do *framework* AspectM para uso da plataforma JADE (Seção 2.6) no contexto de desenvolvimento do sistema EC.

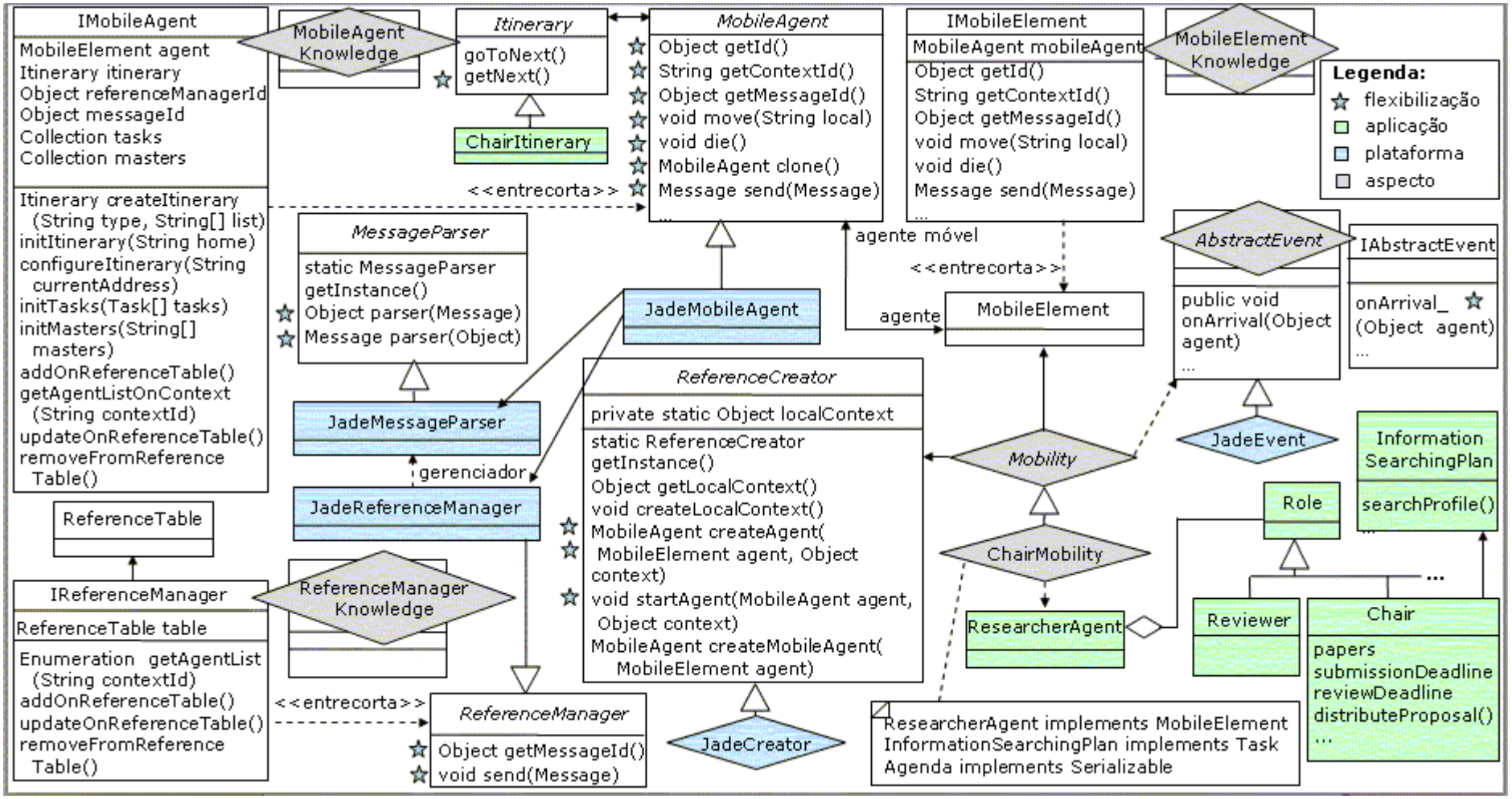

Figura 39. Sistema Expert Committee utilizando AspectM e JADE

Adotamos os seguintes passos no processo de instanciação do AspectM para classes do sistema EC (Seção 6.3.3):

1. para a classe de papel `Chair`, que requer acesso a serviços de mobilidade, criamos um aspecto que estende o aspecto abstrato `Mobility`: o aspecto `ChairMobility`;
2. especificamos no aspecto `ChairMobility`, através de uma declaração intertipo, que um agente `ResearcherAgent` implementa a interface `MobileElement` ("`ResearcherAgent implements MobileElement`" em `ChairMobility`, na Figura 39). Por esta declaração, um agente `ResearcherAgent` torna-se um elemento móvel no sistema;
3. no aspecto `ChairMobility`, através de uma declaração intertipo, especificamos que um plano `InformationSearchingPlan` implementa a interface `Task` ("`InformationSearchingPlan implements Task`" em `ChairMobility`, na Figura 39). Por esta declaração, um plano `InformationSearchingPlan` torna-se uma tarefa que pode ser utilizada na definição do conjunto de junção de movimentação;
4. no aspecto `ChairMobility`, através de uma declaração intertipo, especificamos que um objeto `Agenda` implementa a interface `Serializable` ("`Agenda implements Serializable`" em `ChairMobility`, na Figura 39). Por esta declaração, um objeto `Agenda` pode ser movido junto com um agente `ResearcherAgent`;
5. definimos a classe de itinerário `ChairItinerary` que implementa a interface `Itinerary` do AspectM. A estratégia de seleção do próximo *host* é especificada pela concretização do método `getNext()`;
6. no aspecto `ChairMobility`, especificamos os métodos `getItineraryType()` e `getContextList()`;
7. no aspecto `ChairMobility`, especificamos o conjunto de junção `agentInstantiation()`, que dispara o protocolo de instanciação de `ResearcherAgent` na plataforma de mobilidade JADE;
8. como não existem métodos de aplicação a serem executados durante o protocolo de instanciação de `ResearcherAgent`, não especificamos a execução de operações no método `doAfterAgentInstantiation()` herdado do aspecto `Mobility`;

9. no aspecto `ChairMobility`, especificamos a execução do método `searchProfile()` no plano `InformationSearchingPlan` como o conjunto de junção do protocolo de movimentação de `ResearcherAgent`;
10. no aspecto `ChairMobility`, especificamos o método `searchProfileCheckNecessity()` do protocolo de movimentação. Este método verifica a necessidade de movimentação do agente `ResearcherAgent` no contexto específico de `searchProfile()`;
11. no aspecto `ChairMobility`, especificamos quais os métodos executados no protocolo de inicialização de `ResearcherAgent`. Estes métodos são invocados na concretização do método `doAfterArrivalHost()`, herdado do aspecto `Mobility`;
12. como não existem métodos de aplicação a serem executados no protocolo de destruição de `ResearcherAgent`, não especificamos a execução de operações no método `doBeforeAgentDestruction()` herdado do aspecto `Mobility`.

Por exemplo, a Figura 40 apresenta a definição da classe `ChairItinerary`. `ChairItinerary` implementa a interface `Itinerary` de AspectM a fim de definir a estratégia de seleção do próximo *host* a ser visitado no itinerário de *chair*. A estratégia de seleção é definida através da concretização do método `configureNeighbors()`, invocado no corpo do método `getNext()` da interface `Itinerary`. Através do método `configureNeighbors()`, configuramos dinamicamente os *hosts* acessíveis a partir do *host* corrente no itinerário de *chair*.

No código da Figura 40, a estratégia de seleção implementada pelo método `configureNeighbors()` é trivial: para o *host* corrente (`getIndexOfAddress()`), são escolhidos randomicamente até cinco *hosts* acessíveis (`addNeighboringHost()`).

```
1: public class ChairItinerary extends DynamicItinerary {
2: public ChairItinerary(MobileAgent mobileAgent) {
3:    super(mobileAgent);
4: }
5: ...
6: private int getIndexOfAddress(String address){
7:    int cont=0;
8:    for (Enumeration e = this.addresses();
9:       e.hasMoreElements();){
10:       if (e.nextElement().equals(address)) { break; }
11:       else { cont++; }
12:    }
13:    return cont;
14: }
15: protected void configureNeighbors(String address) {
16:    int index = getIndexOfAddress(address);
17:    int neighbor;
18:    for (int i = 1; i <= 5; i++) {
19:       neighbor = random(this.size(), index);
20:       addNeighboringHost(address,
21:       getAddressAt(neighbor));
22:    }
23: }
24: }
```

Figura 40. Implementação da Classe de Itinerário ChairItinerary

As Figuras 41 a 44 apresentam trechos de código referentes ao protocolo de mobilidade definido no aspecto `ChairMobility`. O aspecto `ChairMobility` estende o aspecto `Mobility` a fim de definir o comportamento de mobilidade específico da classe de papel `Chair`. No aspecto `ChairMobility`, são especificados os pontos de flexibilização de AspectM para o contexto de execução do papel `Chair`. A Figura 41 apresenta as declarações intertipos do aspecto `ChairMobility`, além do protocolo de instanciação de `Chair`.

```
1: public aspect ChairMobility extends Mobility {
2: declare parents: ResearcherAgent implements MobileElement;
3: declare parents: InformationSearchingPlan implements Task;
4: declare parents: Agenda implements Serializable;
5: pointcut agentInstantiation(String referenceName,
6:    MobileElement agent):
7:    this(agent) && args(referenceName,*) &&
8:    initialization(ResearcherAgent+.new(String,*));
9: void doAfterAgentInstantiation(MobileElement agent){ }
10: String getItineraryType(MobileElement agent) {
11:    return "expertcommittee.chair.ChairItinerary";
12: }
13: String[] getContextList(MobileElement agent){...}
14: String[] itinerary = new String[10];
15:    itinerary[0] = "Container-1";
16:    itinerary[1] = "Container-2";
17:    ...
18:    itinerary[9] = "Container-10";
19:    return itinerary;
20: }
21: Task[] getTaskList(MobileElement agent){
22:    Task[] tasks = new Task[1];
23:    InformationSearchingGoal goal = new
24:    InformationSearchingGoal();
25:    InformationSearchingPlan plan = new
26:    InformationSearchingPlan(goal);
27:    tasks[0] = plan;
28:    return tasks;
29: }
30: String[] getMasterList(MobileElement agent) {
31:    return new String[0];
32: }
33: ...
34: }
```

Figura 41. Protocolo de Instanciação no Aspecto ChairMobility

Na Figura 41, o conjunto de junção de instanciação (linhas de 5 a 8) aponta para o tipo de agente `ResearcherAgent` (linha 8) mesmo sendo o protocolo de instanciação específico do papel `Chair`. A definição deste conjunto de junção apontando para o agente `ResearcherAgent` garante que, quando o papel `Chair` se move, `ResearcherAgent` também é movido. Isto é necessário porque as funcionalidades do papel `Chair` dependem do conhecimento intrínseco contido na classe que define o tipo `ResearcherAgent`.

O tipo de itinerário do agente é especificado como sendo a classe `ChairItinerary` no retorno do método `getItineraryType()` (linhas de 10 a 12). A especificação dos *hosts* que compõem o itinerário de `Chair` é efetuada no método `getContextList()` (linhas de 13 a 20). O método `getTaskList` (linhas de 21 a 29) efetua a criação de um plano `InformationSearchingPlan` (linhas 23 a 26). Este plano é incluído na lista de tarefas (linha 27) utilizadas na definição do conjunto de junção de movimentação.

Note que não existem métodos de aplicação executados durante o protocolo de instanciação de `ResearcherAgent`, pois não especificamos a execução de operações no corpo do método `doAfterAgentInstantiation()` (linha 9). O agente `ResearcherAgent` também não especifica a existência de mestres no contexto de execução de seu protocolo de mobilidade (linhas de 30 a 32).

A Figura 42 apresenta a definição do protocolo de movimentação de `Chair`. O conjunto de junção de movimentação é especificado como sendo a execução do método `searchProfile()` (linhas de 3 a 5). O método `searchProfile()` está definido na interface do plano `InformationSearchingPlan` (linha 5). O método `searchProfileCheckNecessity()`, invocado no protocolo de movimentação em `Mobility`, verifica a necessidade de movimentação de `ResearcherAgent` no contexto de execução do método `searchProfile()` (linhas de 6 a 10).

O conjunto de junção `informationNeedChecking()` (linhas de 11 a 13) e o adendo anterior associado (linhas de 14 a 19) são definidos para propósitos específicos de aplicação. Por exemplo, no corpo do adendo `informationNeedChecking()`, quaisquer métodos disponíveis ao tipo de agente `ResearcherAgent` pela "`ResearcherAgent implements MobileElement`" podem ser invocados, inclusive o método `move()` (linha 17).

```
1: public aspect ChairMobility extends Mobility {
2: ...
3: protected pointcut agentMovement(Task task):
4:    this(task) && execution(Hashtable
5:    InformationSearchingPlan.searchProfile(Vector));
6: public boolean
7:    searchProfileCheckNecessity(Task task, Object result){
8:    if (result == null) { return true; }
9:    else { return false; }
10: }
11: pointcut informationNeedChecking(Plan plan):
12:    this(plan) &&
13:    execution(void DistributionPlan.executePlan(..));
14: before (Plan plan): informationNeedChecking(plan) {
15:    ...
16:    ResearcherAgent agent = plan.getAgent();
17:    agent.move("Container-5");
18:    ...
19: }
20: ...
21: }
```

Figura 42. Protocolo de Movimentação no Aspecto ChairMobility

A Figura 43 apresenta a definição do protocolo de inicialização de `Chair`. As operações do protocolo de inicialização são invocadas no corpo do método `doAfterArrivalHost()` (linhas de 3 a 8). Primeiro, um teste para verificar se o agente está ou não em seu *host* original (linha 4). Quando o agente está em um ambiente que não é o seu originalmente, escolhe entre suas tarefas (linhas de 5 a 6) aquela que deve ser executada (linha 7). Lembre-se que na execução do plano `InformationSearchingPlan` (linha 7), se o método `searchProfile()` é executado, então o protocolo de movimentação é disparado no aspecto `Mobility`.

```
1: public aspect ChairMobility extends Mobility {
2: ...
3: protected void doAfterArrivalHost(MobileElement agent) {
4:    if (!agent.isAgentOut()) return;
5:    InformationSearchingPlan plan =(InformationSearchingPlan)
6:    agent.getTaskOfType("InformationSearchingPlan");
7:    plan.executePlan((ResearcherAgent)agent, plan.getGoal());
8: }
9: protected void doAfterReceivingMessage(MobileElement
10:   agent, Message message) { }
11: protected void doBeforeCloneAgent(MobileElement agent) { }
12: protected void doAfterCloneAgent(MobileElement agent) { }
13: protected void doAfterCloneCreation(MobileElement agent) { }
14: ...
15: }
```

Figura 43. Protocolo de Inicialização no Aspecto ChairMobility

Na Figura 43, outros métodos do aspecto `Mobility` (linhas de 9 a 14) podem ser especificados conforme requisitos de sistema. No caso do EC, estes métodos possuem corpo vazio, pois não são utilizados nos cenários de movimentação do papel *chair*. No próximo estudo de caso (Seção 7.3), mostramos como estes métodos podem ser utilizados para os propósitos de tratamento de mensagens recebidas e clonagem de agentes.

A Figura 44 apresenta a definição do protocolo de destruição de `Chair`. Não são especificadas operações que devem ser executadas imediatamente antes à destruição efetiva de um agente (linha 3).

```
1: public aspect ChairMobility extends Mobility {
2: ...
3: protected void doBeforeAgentDestruction(MobileElement agent) {}
4: ...
5: }
```

Figura 44. Protocolo de Destruição no Aspecto ChairMobility

A Figura 45 ilustra um cenário do protocolo de movimentação de `Chair`.

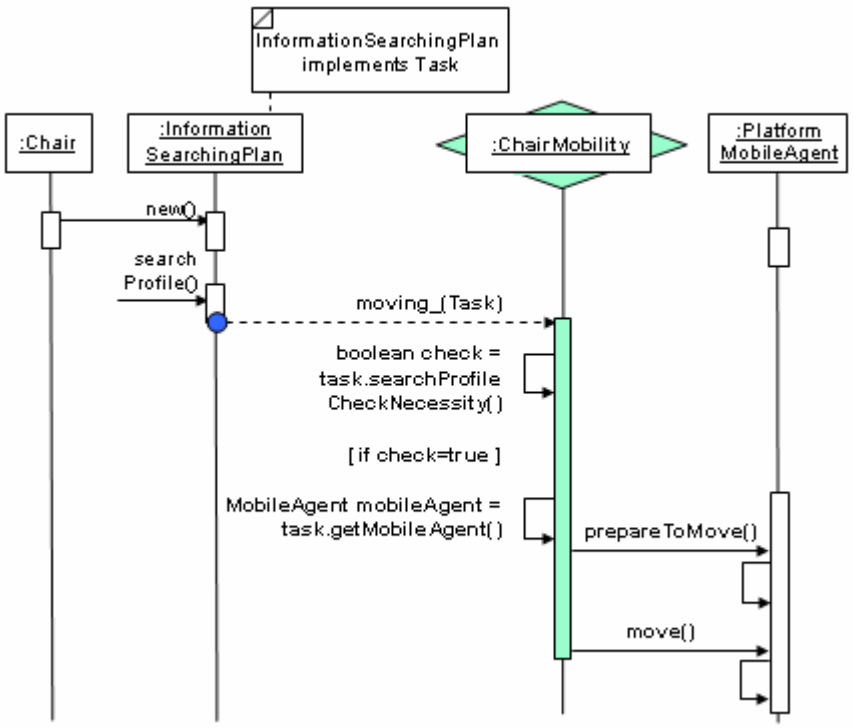

Figura 45. Cenário do Protocolo de Movimentação do Papel Chair

O cenário do protocolo de movimentação ilustrado na Figura 45 corresponde à seguinte seqüência de passos na execução de um plano `SearchingInformationPlan`:

1. instanciação de um objeto `SearchingInformationPlan`, que corresponde à criação de um plano de busca do papel `Chair`. O plano `SearchingInformationPlan` implementa a interface `Task` e, por essa razão, pode ter seu contexto exposto no conjunto de junção de movimentação `agentMovement()`;
2. o aspecto `ChairMobility` detecta o final da execução do método `searchProfile()` no plano `SearchingInformationPlan`, através do conjunto de junção `agentMovement()`;
3. o aspecto `ChairMobility` captura as informações do contexto de execução do plano `SearchingInformationPlan`. Neste caso, a informação relevante é o valor de retorno do método `searchProfile()`;

4. o aspecto `ChairMobility` executa o método `searchProfileCheckNecessity()` a fim de verificar a necessidade de se movimentar o agente desempenhando o papel de *chair*;
5. o aspecto `ChairMobility` invoca o método `prepareToMove()`, a fim de que sejam executadas as últimas ações antes da transferência do agente;
6. o aspecto `ChairMobility` invoca o método `move()`, que delega à plataforma de mobilidade a tarefa de efetivamente movimentar o agente.

Neste ponto, é importante observar que o projeto de EC permanece inalterado mesmo quando optamos pela utilização de outra plataforma cujo pacote de comunicação com o AspectM já tenha sido desenvolvido. O *framework* AspectM garante o uso flexível de plataformas de mobilidade. A Figura 46 apresenta o projeto do sistema *Expert Committee* usando *Aglets*.

Na Figura 46, note que, embora as classes de plataforma se refiram a *Aglets*, as classes do sistema EC permanecem com a mesma implementação. Isto também ocorre com qualquer outra aplicação que utiliza o *framework* AspectM optando por serviços de plataformas cujos pacotes de comunicação com o AspectM já estejam disponíveis.

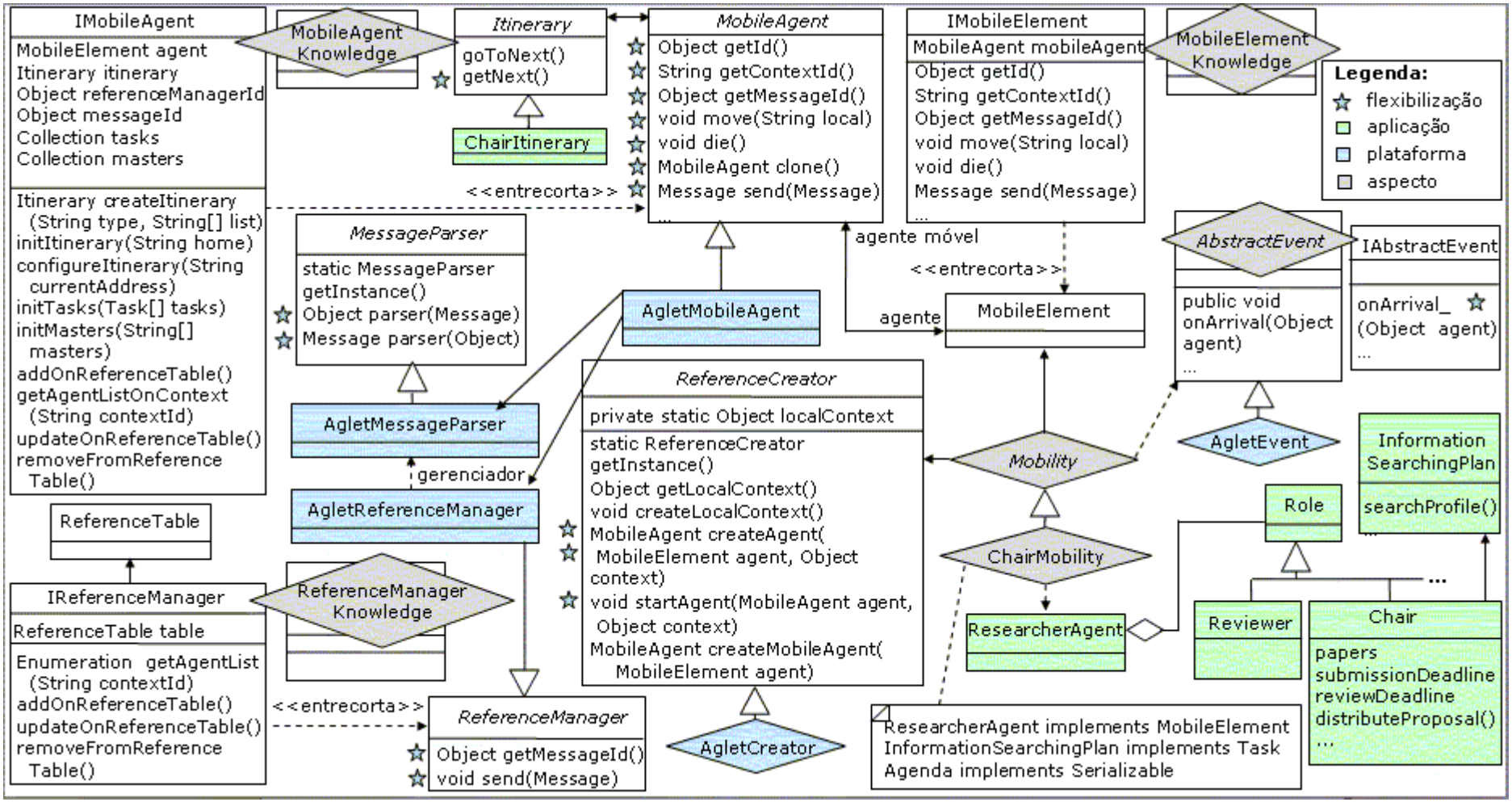

Figura 46. Sistema Expert Committee utilizando AspectM e Aglets

7.2. MobiGrid Framework

MobiGrid (Barbosa & Goldman, 2005) consiste em um *framework* baseado em agentes móveis utilizados em um ambiente de grade computacional, denominado *InteGrade* (Goldchleger et al., 2004). O objetivo de *MobiGrid* é permitir que tarefas seqüenciais longas possam ser executadas em ciclos de CPU ociosos de estações pessoais de trabalho em uma rede (Barbosa & Goldman, 2005). Neste contexto, os agentes móveis podem ser usados para encapsular tarefas que requerem longo tempo de processamento, pois possuem a capacidade de migrar sempre que uma máquina é requisitada por seu usuário local.

7.2.1. Motivação

O paradigma de grades computacionais (Foster & Kesselman, 1999) foi proposto a fim de resolver os dois maiores problemas do uso de aglomerados[25] e supercomputadores: preços altos e desperdício de recursos. Em uma grade computacional, vários computadores pessoais interconectados por uma rede são utilizados para prover maior poder computacional à execução de tarefas. Uma grande vantagem da grade é que seu conceito enfrenta diretamente o problema do desperdício, tendo em vista que sempre que um computador está ocioso, seu poder computacional pode ser disponibilizado para a grade. Além disso, outra grande vantagem da grade é seu custo reduzido, visto que podemos constituí-la somente com computadores já disponíveis conectados a uma rede, utilizando para isso, ferramentas de software livre.

Nesse contexto, a idéia de agentes móveis é interessante. Eles podem ser usados para encapsular aplicações oportunistas, as quais podem usar o pouco tempo computacional eventualmente disponível em estações de trabalho, migrando para outra máquina sempre que o usuário local solicita sua máquina e preservando sempre o processamento já realizado. Dessa maneira, agentes móveis podem ser considerados como uma ferramenta complementar para diminuir ainda mais o tempo ocioso de uma grade.

25 Do inglês: *clusters*.

A capacidade de migração de agentes móveis atende a dois requisitos primordiais do projeto *InteGrade* (Barbosa & Goldman, 2005): (i) a presença do sistema deve ser transparente aos usuários das máquinas, isto é, os usuários devem ter prioridade com relação às aplicações do *InteGrade*; (ii) os recursos ociosos devem ser utilizados da melhor forma possível. No caso da necessidade de liberação dos recursos da máquina para o usuário local, por exemplo, se o usuário acessa uma máquina que está rodando código móvel, esse código pode migrar para outra máquina sem perder o processamento já realizado.

A arquitetura do *InteGrade* fornece os dados sobre o estado da rede e das máquinas, permitindo que agentes móveis possam escolher máquinas com recursos computacionais mais adequados e disponíveis (Barbosa & Goldman, 2005). Para esta decisão, padrões de utilização de recursos das outras máquinas podem ser utilizados. Dada a característica oportunista dos agentes móveis, sua utilização fica muito indicada para melhor utilização de recursos ociosos, já que esses agentes podem utilizar até períodos curtos de ociosidade, caminhando no sentido da ociosidade próxima de zero almejada pelo *InteGrade* (Barbosa & Goldman, 2005).

A possibilidade de migração para uma máquina com recursos computacionais mais poderosos, caso alguma máquina com recursos melhores tenha ficado ociosa, é muito interessante, exceção feita à possibilidade de o padrão de comportamento dela indicar que sua ociosidade não durará muito tempo. Neste contexto, os agentes móveis seriam utilizados de forma complementar às aplicações do *InteGrade*, permitindo um melhor aproveitamento dos recursos computacionais. A idéia principal do *framework MobiGrid* é fornecer ao desenvolvedor um ambiente para a programação de tarefas longas. O *framework* foi implementado sobre *Aglets* (Barbosa & Goldman, 2005).

7.2.2. Arquitetura Geral

A seguir, são descritos em alto nível os principais componentes do *framework MobiGrid* (Barbosa & Goldman, 2005):

- *tarefas*: aplicações longas, encapsuladas dentro de um agente móvel. Na implementação dessas tarefas, o programador deve ter a preocupação de

salvar o estado da tarefa de tempos em tempos, visto que as plataformas de mobilidade Java fornecem mecanismos que suportam apenas a migração fraca. Para ajudar o programador, o *framework MobiGrid* oferece um método especial usado para assinalar que a tarefa alcançou um estado consistente, estando pronta para migração ou clonagem;

- *gerente*: o componente responsável pelo registro das tarefas. Deve estar sempre ativo na máquina do usuário que submeteu uma tarefa ao *framework*. As duas principais funcionalidades do gerente são:
    a. migração, quando da submissão da tarefa, esse gerente consulta a infra-estrutura do *InteGrade* procurando por uma máquina que está ociosa e tem certa probabilidade de permanecer nesse estado por algum tempo. De posse dessa informação, o gerente despacha a tarefa para tal máquina onde ela passa a ser executada. Por outro lado, quando a tarefa necessita migrar, consulta o gerente para obter o endereço de uma máquina ociosa. O gerente obtém essa informação utilizando a infra-estrutura do *InteGrade*;
    b. vivacidade, o gerente se encarrega de criar um clone dessa tarefa e o despacha para uma máquina distinta do primeiro, para garantir a vivacidade[26] da mesma. O nível de vivacidade indica o nível de qualidade de uma tarefa sendo executada em alguma máquina. Quando da morte de um dos gêmeos – que pode ocorrer por vários motivos, como, por exemplo, uma interrupção de energia inesperada na máquina onde a tarefa está rodando – o gerente clona o gêmeo ainda vivo e o despacha para outra máquina. A palavra "gêmeos" é utiliza para referenciar as cópias de uma tarefa;
- *servidores leves*: servidor instalado em cada uma das máquinas que oferece recursos ao *framework*. Oferece um ambiente para execução das tarefas. Quando da solicitação da máquina por seu usuário local, o servidor leve solicita a evacuação das tarefas hospedadas por ele. Essas tarefas consultam seus gerentes, os quais se comunicam com o InteGrade na procura de máquinas ociosas. Quando obtêm tal

26 Do inglês: *liveness*.

informação, os gerentes providenciam a evacuação dessas tarefas para novas máquinas;

- *daemon*: encarrega-se de verificar se a máquina está ociosa ou não. Quando ociosa, ele avisa o *InteGrade* e levanta o servidor leve, para o recebimento de tarefas. Quando da requisição da máquina por parte do seu usuário, o *daemon* informa ao servidor, o qual efetua a evacuação das suas tarefas, sendo, em seguida, desligado pelo *daemon*;
- *cliente*: componente utilizando para a submissão de tarefas ao *framework*. Também oferece um ambiente de hospedagem ao gerente.

A Figura 47 ilustra a arquitetura geral do *framework MobiGrid*. Na Figura 47, note que cada um dos clientes hospeda um gerente, o qual se comunica com o *InteGrade*. O gerente do Cliente 1 gerencia a Tarefa 1 e sua cópia; o gerente do Cliente 2 gerencia a Tarefa 2 e sua cópia. Note também que os *daemons* comunicam-se com o *InteGrade* para avisá-lo quando a máquina está ociosa, disparando então a execução do servidor leve.

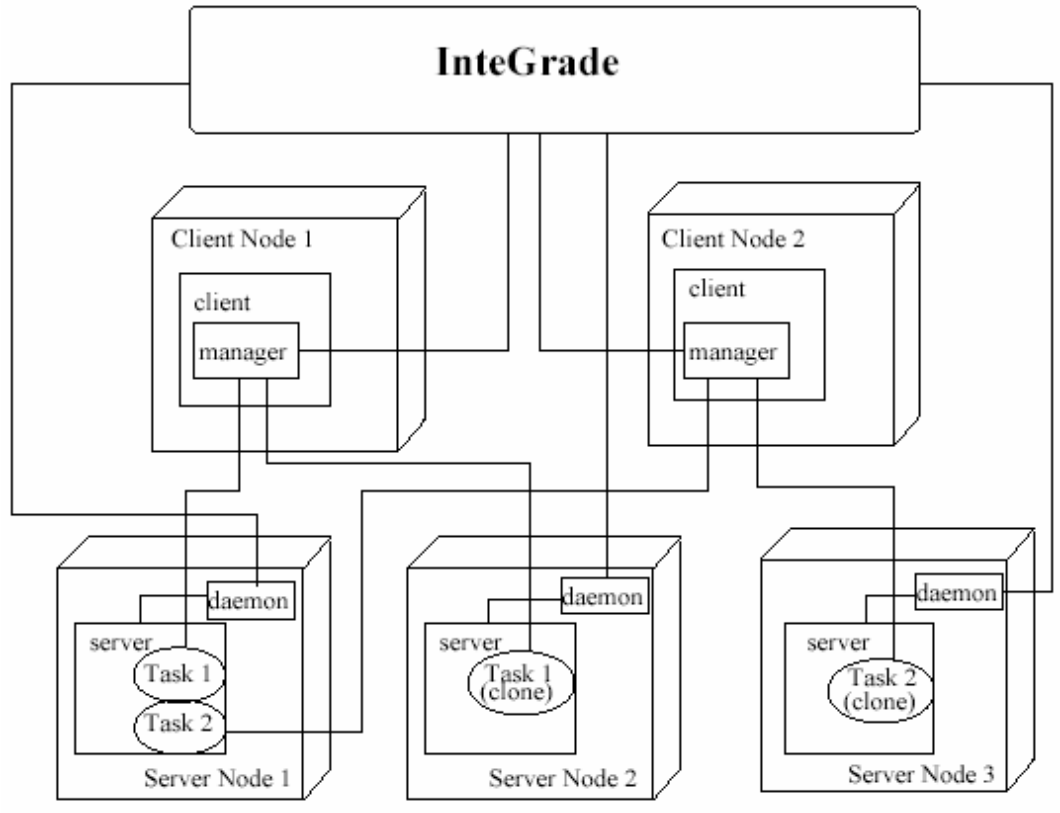

Figura 47. Arquitetura Geral do Framework MobiGrid

7.2.3.
Reengenharia para Uso de AspectM

A Figura 48 apresenta o diagrama de classes do *framework MobiGrid*.

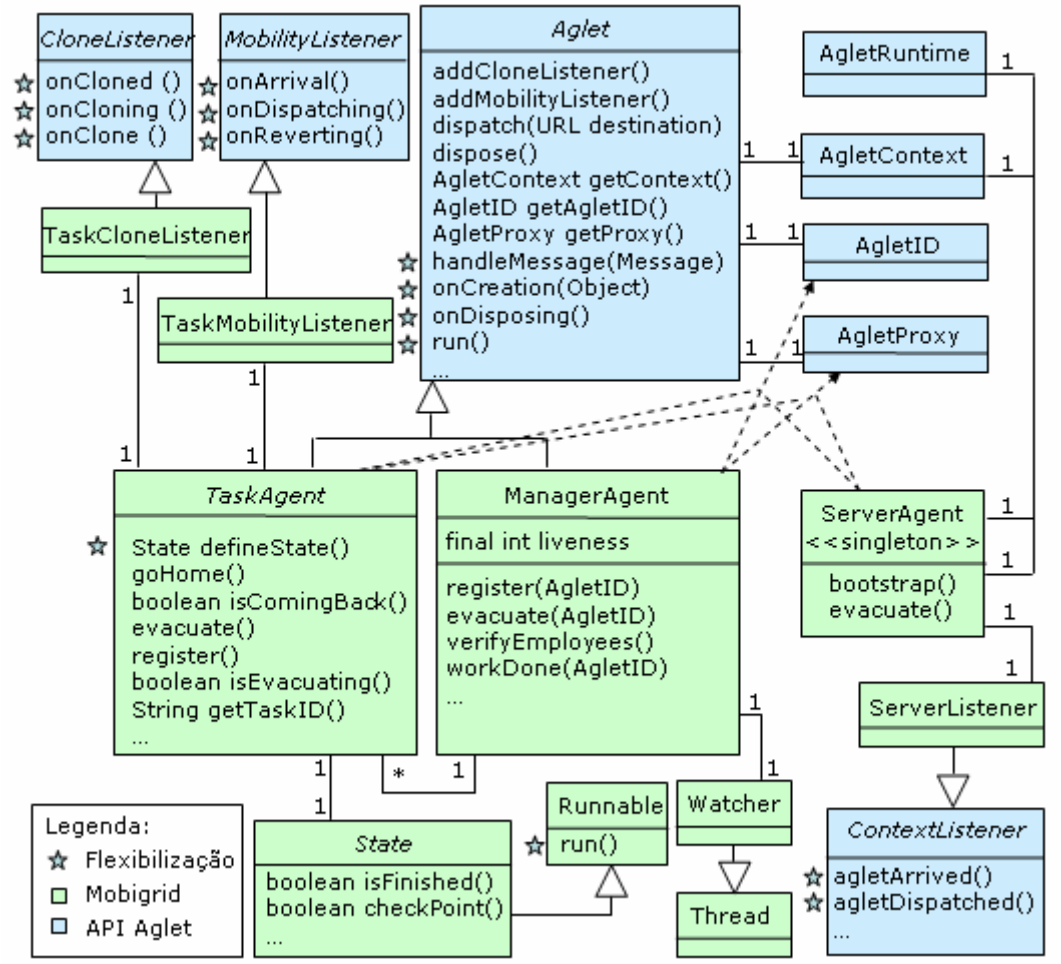

Figura 48. Diagrama de Classes do Framework MobiGrid

Para realizarmos a reengenharia do *framework MobiGrid* utilizando AspectM, identificamos os seguintes requisitos de projeto após análise do diagrama de classes da Figura 48:

- **reimplementação dos servidores leves**: o servidor (instância da classe `ServerAgent`) instalado em cada uma das máquinas que oferece recursos ao *framework* é implementado como uma classe que faz referência direta a elementos internos do *runtime* de *Aglets*. Por outro lado, a AAPI (Seção 2.6.1) não dispõe de uma interface que permite a manipulação de contextos e agentes, sem o conhecimento de elementos internos ao *runtime* da plataforma. Além disso, uma reimplementação

dos servidores leves deve *manter* sua função original de envio de mensagens de evacuação após buscas em contextos de execução;

- **disponibilização de observadores de eventos**: os servidores e as tarefas (instâncias da classe `TaskAgent`) necessitam executar ações em momentos específicos do ciclo de vida de agentes móveis (Seção 2.2). Por esta razão, é necessário possibilitar a especificação de procedimentos que devem ser executados *antes* e/ou *após* a execução de operações como clonagem, movimentação e troca de mensagens. Entretanto, a disponibilização de observadores de eventos não deve manter um alto nível de acoplamento entre o projeto de *MobiGrid* e o modelo de eventos de *Aglets*;
- **reestruturação do projeto quanto à manipulação de contextos**: os servidores, as tarefas e os gerentes (instâncias da classe `ManagerAgent`) efetuam buscas por agentes em seus respectivos contextos de execução. Entretanto, como o conceito de contextos tal como implementado em *Aglets* não é recorrente em outras plataformas de mobilidade, é necessário haver a reestruturação do projeto de *MobiGrid* no que tange aos serviços que manipulam *diretamente* os contextos de execução providos por servidores leves;
- **reestruturação do projeto quanto à manipulação de identificadores para troca de mensagens**: os servidores, as tarefas e os gerenciadores efetuam trocas de mensagens em cenários que envolvem registro de novas tarefas, evacuação de tarefas em máquinas que são solicitadas por seus usuários, clonagem de tarefas por seus respectivos gerenciadores, movimentação de tarefas para outros *hosts* disponíveis na grade. Entretanto, como o mecanismo de comunicação de *Aglets* é baseado em *proxies*, é necessário haver a reestruturação do projeto de *MobiGrid* no que se refere à identificação dos receptores de mensagens.

A Figura 49 apresenta o resultado da reengenharia de *MobiGrid* usando o *framework* AspectM.

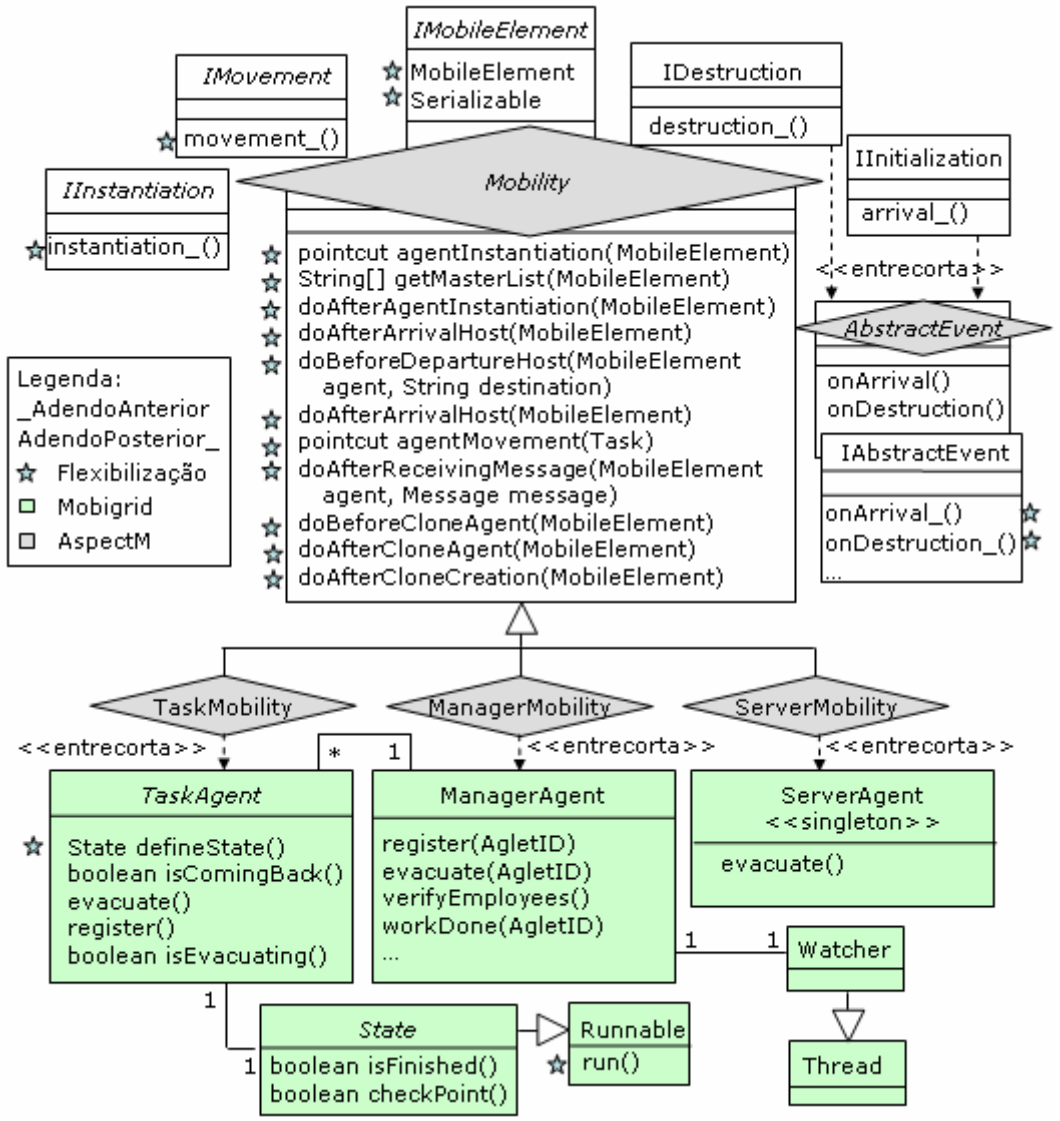

Figura 49. Reengenharia do Framework MobiGrid utilizando AspectM

Na Figura 49, ao executarmos a reengenharia de *MobiGrid* utilizando AspectM, efetuamos várias modificações no projeto da Figura 48. As principais modificações realizadas na reengenharia de *MobiGrid* são descritas a seguir.

Desacoplamento de servidores leves do *runtime* de *Aglets*. A classe `ServerAgent` é reimplementada para manter *apenas* sua função original de envio de mensagens solicitando a evacuação de tarefas. Como o AspectM encapsula a manipulação dos elementos internos de *Aglets*, não é mais necessário efetuar referências diretas ao *runtime* da plataforma. Por exemplo, no projeto da Figura 48, o método `bootstrap()` da classe `ServerAgent` manipula instâncias das classes `AgletContext` e `AgletRuntime`. No projeto da Figura 49, a classe `ServerAgent` mantém apenas o método `evacuate()`.

Reprojeto das classes de observadores de eventos. Como o aspecto `Mobility` disponibiliza pontos de flexibilização onde é possível especificar procedimentos a serem executados *antes* e/ou *após* a execução de operações como clonagem, movimentação e troca de mensagens, os procedimentos invocados nos métodos das classes `TaskCloneListener`, `TaskMobilityListener` e `ServerListener` são transferidos para o corpo dos métodos correspondentes no aspecto `TaskMobility`. Com o AspectM, ao utilizarmos observadores de eventos, diminuímos o nível de acoplamento entre SMAs e o modelo de eventos de uma plataforma de mobilidade.

Remoção de referências diretas a contextos. Em geral, a manipulação de um contexto em *Aglets* é necessária apenas como passo intermediário na busca por identificadores de agentes para troca de mensagens ou no armazenamento temporário de dados para uso posterior de agentes em aplicações. Na busca por identificadores, a manipulação de contextos pode ser realizada por agentes que encapsulam serviços de gerenciamento de contexto (Seção 6.4.1). Os gerenciadores de contexto respondem a solicitações comuns de buscas por agentes em contextos de execução. Em cenários de armazenamento temporário de dados, outra estratégia pode ser facilmente implementada pelas aplicações, de modo que, também através de troca de mensagens, os dados possam se tornar acessíveis aos agentes interessados. No projeto original de *MobiGrid*, um contexto é manipulado somente para a execução da tarefa de busca por agentes. Com o uso de AspectM, as referências diretas a contextos são facilmente reimplementadas através de invocações ao método `getAgentListOnContext()`. Este método está disponível para os todos os agentes que declaram a implementação da interface `MobileElement` (Seção 6.3.3). Portanto, porções do código de *MobiGrid* com invocações ao método `getAgletContext().getAgletProxies()` são substituídas por chamadas a `getAgentListOnContext(String)`, que são solicitações tratadas pelo `ReferenceManager` (Seção 6.4.2) de AspectM.

Remoção de referências diretas a *proxies*. A manipulação de *proxies* de *Aglets* é utilizada no processo de troca de mensagens ou em solicitações de agentes mestres a escravos (Seção 3.3) para a execução de procedimentos do protocolo de mobilidade (Seção 2.6). Entretanto, a manipulação de *proxies* pode ser efetuada, assim como a de contextos, por agentes que encapsulam serviços de

gerenciamento de contexto (Seção 6.4.1). Usando AspectM, os *proxies* são substituídos por descritores de agentes retornados por métodos como `getAgentListOnContext()`. Mais especificamente, porções do código de *MobiGrid* com invocações a métodos para envio de mensagens, como `proxy.sendMessage()`, são substituídas por métodos correspondentes na interface `MobileElement`, como `sendMessage(Message)`, sem referências a *proxies*. Na formatação da mensagem passada como argumento no método `sendMessage(Message)`, deve ser especificado o descritor que corresponde ao agente receptor da mensagem. Este descritor pode ser obtido através de uma busca no retorno de `getAgentListOnContext()`. Um agente específico pode ser identificado pela aplicação do método `getReferenceName()` sobre o descritor, sendo que o atributo `referenceName` é configurado previamente no conjunto de junção de instanciação do agente (Seção 6.3.3). O procedimento de remoção de referências diretas a *proxies*, é análogo para invocações a operações do protocolo de mobilidade, como `proxy.dispatch()`, as quais são substituídas por mensagens com formato pré-estabelecido para operações de clonagem, movimentação e destruição de agentes. Como um descritor possui um identificador imutável em todo o ciclo de vida de um agente e este identificador é configurado pela própria aplicação, o uso direto de *proxies* torna-se desnecessário e, desta forma, a flexibilização de plataformas é mantida.

As Figuras 50 a 52 apresentam trechos de código dos aspectos `ServerMobility`, `ManagerMobility` e `TaskMobility`. Na Figura 50, note a especificação do conjunto de junção de instanciação do representante de `ServerAgent` na plataforma (linhas de 3 a 6). O representante é instanciado para possibilitar ao agente `ServerAgent` o envio de mensagens de evacuação através de métodos disponíveis na interface `MobileElement` (linha 2).

```
1: public aspect ServerMobility extends Mobility {
2: declare parents: ServerAgent implements MobileElement;
3: pointcut agentInstantiation(String referenceName,
4:    MobileElement agent): this(agent) &&
5:    args(referenceName,*) &&
6:    initialization(ServerAgent+.new(String,*));
7: void doAfterAgentInstantiation(MobileElement agent){ }
8: ...
9: }
```

Figura 50. Reengenharia de MobiGrid: Aspecto ServerMobility

Na Figura 51, especificamos o método `doAfterReceivingMessage()` (linhas de 9 a 13), que possibilita o tratamento de mensagens recebidas pelo agente `ManagerAgent`. Em outras palavras, o representante de um agente `ManagerAgent` na plataforma é instanciado para permitir que `ManagerAgent` troque mensagens com outros agentes, especialmente com as tarefas sendo executadas sob seu gerenciamento.

```
1: public aspect ManagerMobility extends Mobility {
2: declare parents: ManagerAgent implements MobileElement;
3: pointcut agentInstantiation(String referenceName,
4:    MobileElement agent):
5:    this(agent) && args(referenceName,*) &&
6:    initialization(ManagerAgent+.new(String,*));
7: void doAfterAgentInstantiation(MobileElement agent){ }
8: ...
9: void doAfterReceivingMessage(MobileElement agent,
10:   Message message){
11:   ManagerAgent manager = (ManagerAgent) agent;
12:   manager.handleMessage(message);
13: }
14: ...
15: }
```

Figura 51. Reengenharia de MobiGrid: Aspecto ManagerMobility

Na Figura 52, em `TaskMobility`, vários métodos são concretizados para a definição do comportamento de um agente `TaskAgent`. Ao contrário do que ocorre com agentes `ServerAgent` e `ManagerAgent`, o representante de um agente `TaskAgent` é instanciado para efetivamente obter acesso aos serviços de mobilidade de plataformas.

```
1: public aspect TaskMobility extends Mobility {
2: declare parents: TaskAgent implements MobileElement;
3: pointcut agentInstantiation(String referenceName,
4:    MobileElement agent): this(agent) &&
5:    args(referenceName,*) &&
6:    initialization(TaskAgent+.new(String,*));
7: void doAfterAgentInstantiation(MobileElement agent){...}
8: String getItineraryType(MobileElement agent)
9:    { return null; }
10: String[] getItineraryList(MobileElement agent)
11:    { return null; }
12: Task[] getTaskList(MobileElement agent)
13:    { return null; }
14: String[] getMasterList(MobileElement agent){
15:    Enumeration idList = agent.
16:    getAgentListOnTheContext(agent.getCurrentAddress());
17:    /* busca por descritor do manager */
18:    String[] masters = new String[1];
19:    masters[1] = managerName;
20:    return masters;
21: }
22: pointcut agentMovement(Task task);
23: void doAfterReceivingMessage(MobileElement agent,
24:    Message message){
25:    TaskAgent task = (TaskAgent) agent;
26:    task.handleMessage(message);
27: }
28: ...
29: }
```

Figura 52. Reengenharia de MobiGrid: Aspecto TaskMobility

Na Figura 52, uma observação importante é que, embora uma instância de `TaskAgent` seja efetivamente um agente móvel, seus métodos referentes à configuração de itinerário não são concretizados (linhas de 8 a 11). Isto ocorre porque a seleção do próximo *host* de `TaskAgent` é determinada pela própria infra-estrutura de *InteGrade* (Barbosa & Goldman, 2005). Outro detalhe interessante é a não-concretização do conjunto de junção de movimentação (linha 20). Isto ocorre porque o movimento de `TaskAgent` é disparado em momentos não previamente estabelecidos de seu ciclo de vida.

7.3. Análise da Arquitetura ArchM e do *Framework* AspectM

No *framework* AspectM, aspectos são usados como abstrações capazes de capturar de forma modular a mobilidade dos agentes nas aplicações. Nesta seção, analisamos as soluções fornecidas pelo *framework* AspectM para os problemas de espalhamento e entrelaçamento de código de mobilidade descritos no Capítulo 3.

Através do uso direto da API ou *framework* OO, as várias classes correspondentes a tipos e/ou papéis de agentes de SMAs móveis devem estender a classe base para criação de agentes em uma plataforma. Por exemplo, para possuir a capacidade de mobilidade na plataforma JADE, um agente deve estender a classe `JADEAgent` (Seção 2.6). O uso do mecanismo de herança nesta situação resulta em replicação e entrelaçamento de código: a mobilidade do agente está implementada em meio às funcionalidades básicas e às atividades de colaboração do agente (problema ① da Seção 3.2).

Para resolver este problema, o *framework* AspectM detecta a execução do construtor do tipo de agente ao qual deve ser adicionada a capacidade de mobilidade através do conjunto de junção `agentInstantiation()`. O conjunto de junção `agentInstantiation()` é especificado pelo usuário no subaspecto de `Mobility` correspondente ao agente móvel instanciado. No adendo correspondente ao conjunto de junção `agentInstantiation()`, o aspecto `Mobility` implementa o protocolo de instanciação do agente na plataforma.

`MobileElement` constitui a interface do *framework* AspectM que permite a interação entre os agentes de uma aplicação e as classes de plataforma que disponibilizam os serviços de mobilidade aos SMAs móveis. Os usuários de

AspectM especificam nos subaspectos de `Mobility` quais tipos de agentes implementam esta interface e, por esta declaração, os serviços de mobilidade tornam-se acessíveis aos agentes nas aplicações. Por exemplo, atributos e métodos de manipulação de itinerário estão encapsulados no *framework* AspectM, mas a manutenção e configuração destes pode ser efetuada nos subaspectos de `Mobility` por meio de invocações a métodos providos pela interface `MobileElement`.

Portanto, uma conseqüência do uso da interface `MobileElement` é a resolução dos problemas ② e ③ da Seção 2.3: através do uso conjunto de APIs/*frameworks* OO de plataformas de mobilidade e padrões de projeto, as classes referentes aos papéis e tipos de agentes também precisam manter atributos e métodos para o tratamento de questões específicas de mobilidade – como, por exemplo, o atributo `itinerary` (problema ②) e métodos que efetuam a manutenção destes atributos (problema ③); com a utilização de `MobileElement` em conjunto com o padrão *Itinerary* (Seção 3.3), torna-se possível manter o encapsulamento de atributos e métodos associados ao itinerário, de maneira que estes interesses permaneçam separados do código de agentes nas aplicações.

Na Seção 2.3, vimos que vários métodos de SMAs possuem código de mobilidade a fim de decidir se o agente deve ou não se movimentar para um *host* remoto (problema ④ da Seção 2.3), ou ainda, para decidir se o agente deve ou não retornar ao *host* original (problema ⑤ da Seção 2.3). Os interesses de mobilidade encontram-se replicados em vários pontos do código e espalhados no corpo de métodos de vários planos, papéis e tipos de agente.

No *framework* AspectM, estes problemas são resolvidos através da especificação de conjuntos de junção e métodos abstratos nos subaspectos de `Mobility`. Através do conjunto de junção `agentMovement()`, o usuário especifica o instante em que pode haver a necessidade de movimentação. Além disso, o usuário expõe o contexto de execução do ponto de junção especificado em `agentMovement()` para que o método `checkMobilityNeed()` verifique a necessidade de movimentação por parte do agente. Este método é executado no corpo do adendo correspondente ao protocolo de movimentação e é implementado pelos usuários nos subaspectos de `Mobility`. Portanto, pela implementação do protocolo de movimentação de AspectM, a decisão de movimentação, bem como

as chamadas aos métodos de mobilidade de uma plataforma, permanecem isoladas das funcionalidades básicas e outros interesses de agentes em aplicações.

Por outro lado, pode haver ainda o espalhamento de pré e pós-condições que usualmente devem ser satisfeitas quando um determinado agente se move para outro ambiente (problema ⑥ da Seção 2.3). Por exemplo, logo após a chegada de um agente em um novo *host*, é usual o envio de notificações de chegada. O envio de tal notificação é um procedimento praticamente independente de aplicação, uma vez que pelos menos os gerenciadores de contexto de plataformas (Seção 6.4.1) devem ter seus dados atualizados quanto da chegada de um agente em um novo *host*.

O *framework* AspectM resolve estes problemas pela própria implementação do protocolo de mobilidade de SMAs móveis. Particularmente, o envio de mensagens de chegada a gerenciadores de contexto é uma funcionalidade encapsulada no protocolo de movimentação do *framework*. No AspectM, novamente os interesses de mobilidade, como a comunicação com gerenciadores de contexto, permanecem isolados do código de agentes nas aplicações.

Em SMAs móveis, existe ainda a necessidade de que várias classes declarem a implementação de interfaces provenientes de plataformas de mobilidade. Por exemplo, a declaração de que classes implementam a interface `Serializable` é necessária para que os objetos instanciados a partir destas possam ser movidos junto com os agentes (problema ⑥ da Seção 2.3). Este problema não se restringe à interface `Serializable`. As APIs OO das plataformas de mobilidade geralmente disponibilizam várias interfaces com métodos que devem ser implementados por classes dos sistemas a fim de que planos e ações de um agente possam ser executados automaticamente em momentos específicos do ciclo de vida dos agentes (Seção 2.6).

No *framework* AspectM, este problema é resolvido através de declarações intertipos. Por exemplo: no aspecto abstrato de mobilidade, declara-se que a classe abstrata `Itinerary` é serializável, pois o atributo itinerário deve sempre ser transferido junto com o agente móvel, independentemente da aplicação instanciada; por outro lado, no sistema EC (Seção 3.1), a classe `Agenda` deve ser declarada como elemento serializável no aspecto concreto de mobilidade, uma vez que o papel `Chair` precisa transferir consigo o objeto correspondente à agenda de

um usuário. A transferência de `Agenda` juntamente com `Chair` é um requisito específico de aplicação, e por esta razão, a declaração de que a classe `Agenda` é serializável deve ser realizada nos aspectos concretos de mobilidade. Mesmo assim, note que a declaração de implementação da interface Serializable não aparece espalhada em SMAs, pois todos os elementos serializáveis podem ser especificados em um único no subaspecto de `Mobility`.

As restrições impostas pelas APIs OO de plataformas de mobilidade também introduzem problemas conceituais na especificação de SMAs (Seção 3.2). Um primeiro exemplo envolve a classe `Agent`, superclasse de todos os agentes de um sistema. A classe `Agent` deve estender a classe base para criação de agentes móveis em uma plataforma, mesmo quando determinados tipos de agentes na hierarquia de classes são estacionários (Seção 3.2). O problema é que não é possível definir que uma subclasse de `Agent` estende diretamente a classe base para criação de agentes da plataforma, porque a maioria das linguagens de programação OO não provê suporte à definição de herança múltipla.

Com aspectos, o *framework* AspectM possibilita definir que uma subclasse de `Agent` possui um representante na plataforma de mobilidade (solução para o problema ①), sem que seja necessário efetuar modificações na hierarquia de herança de SMAs. Desta forma, pela utilização de AspectM, não é necessário mais especificar que `Agent` é subclasse de classes pertencentes a plataformas.

Por fim, vimos que a introdução de mobilidade pode requerer modificações no projeto e na implementação de SMAs, como renomeação de métodos e conseqüentes mudanças nas chamadas a estes (Seção 3.2). Da mesma forma que para o problema na hierarquia de `Agent`, a solução para este problema é alcançada, porque a especificação de relacionamentos entre classes não mais é realizada somente através de delegação e herança, mas através de abstrações e mecanismos OA. Pelo encapsulamento dos serviços de mobilidade no *framework* AspectM, torna-se possível especificar que a classe `Agent` define um método `getName()`, mesmo que este método possua uma finalidade diferente de um método com mesmo nome na plataforma de agentes.

Na Tabela 3, apontamos os requisitos de mobilidade (Seção 3.5) que foram considerados nos estudos de caso *Expert Committee* e *MobiGrid*.

Tabela 3. Nível de Satisfação de Requisitos de Mobilidade por AspectM

Requisitos x Aplicação em Estudos de Caso	Expert Committee	MobiGrid
P1. Introdução de mobilidade em agentes estacionários.	Nenhum cenário.	Nenhum cenário.
P2. Flexibilização do uso de plataformas de mobilidade.	Satisfatório. Instanciação para JADE e Aglets.	Satisfatório. Instanciação para JADE e Aglets.
P3. Em cenários de papéis, agentes associados devem ser movidos.	Satisfatório. Papel Chair móvel, ResearcherAgent também móvel.	Nenhum cenário.
A1. Possibilidade de especificar elementos que são móveis (tipo, papéis, tarefas de agentes).	Satisfatório. Tipo Móvel: ResearcherAgent; Papel Móvel: Chair; Tarefa: InformationSearchingPlan.	Satisfatório. Tipos Móveis: ServerAgent, ManagerAgent e TaskAgent.
A2. Possibilidade de especificar pontos de instanciação de agentes móveis na plataforma.	Satisfatório. Ponto de Instanciação: execução do construtor ResearcherAgent().	Satisfatório. Ponto de Instanciação: execução dos construtores ServerAgent(), ManagerAgent() e TaskAgent() .
A3. Encapsulamento de parte do protocolo de instanciação, como inicialização de itinerário, tarefas e mestres do agente.	Satisfatório. Para Chair, inicialização de itinerário, especificação de estratégia de seleção do próximo host e definição de tarefas.	Satisfatório. Para TaskAgent, inicialização de mestres.
A4. Possibilidade de especificar pontos de decisão de movimento.	Satisfatório. Ponto de Movimento: execução do método searchProfile() no plano InformationSearchingPlan.	Nenhum cenário.
A5. Procedimentos invocados no protocolo de movimentação, como finalização de processos e envio de mensagens.	Satisfatório. Na partida de Chair, envio de mensagem de partida.	Satisfatório. No cenário de Evacuação de TaskAgent, finalização de processos e envio de mensagem.
A6. Encapsulamento de parte do protocolo de inicialização, como o envio de mensagem e o acesso à lista de agentes.	Satisfatório. Para Chair, envio de mensagem de chegada e acesso à lista de agentes.	Satisfatório. Para TaskAgent, envio de mensagem de chegada e acesso à lista de agentes.
A7. Especificação de circunstâncias excepcionais nas quais um agente deve retornar.	Nenhum cenário.	Nenhum cenário.
A8. Procedimentos invocados no protocolo de destruição, como finalização de processos.	Nenhum cenário.	Nenhum cenário.
T1. Especificação de objetos serializáveis.	Em Chair, seus atributos não-primitivos internos.	Nenhum cenário.

Observe na Tabela 3 que alguns requisitos de SMAs móveis não foram avaliados empiricamente, uma vez que não figuravam em quaisquer cenários de mobilidade das aplicações *Expert Committe* e *Mobigrid*. Por exemplo, a introdução de mobilidade em agentes estacionários (P1), a invocação de procedimentos excepcionais de retorno (A7) e do protocolo de destruição (A8) são

requisitos não avaliados em nossos estudos (Tabela 3). Entretanto, pela experiência com o uso do *framework* AspectM, podemos afirmar que:

- é possível a introdução transparente de mobilidade em agentes que no projeto de aplicações são inicialmente especificados como estacionários, uma vez que a introdução dos interesses de mobilidade nestes agentes corresponde apenas à tarefa de concretização do aspecto abstrato `Mobility` para estes agentes. Acreditamos que, com a utilização diligente das abstrações e os mecanismos OA, o resultado da tarefa de introduzir ou retirar o aspecto concreto de `Mobility` não tem impacto no funcionamento geral do sistema. Eventualmente, pode haver cenários em que o projeto das aplicações deva ser revisitado, mas isto em geral ocorre pela definição, exposição ou utilização incorreta de pontos de junção nas aplicações;
- a invocação de procedimentos de retorno e de destruição é tratada de forma adequada, uma vez que o projeto destas funcionalidades no *framework* AspectM segue o padrão geral dos protocolos de instanciação e inicialização de agentes.

Como o *framework* AspectM é desenvolvido segundo as diretivas de ArchM, temos que a aplicabilidade do *framework* nos estudos de caso deste trabalho corresponde também à aplicabilidade de nossa arquitetura. Em nosso ponto-de-vista, os dados da Tabela 3 demonstram que o *framework* AspectM satisfaz a vários requisitos importantes de SMAs móveis. Uma avaliação mais rigorosa do tratamento dado a estes e aos outros requisitos não tratados será tema de trabalhos futuros.