6
O *Framework* AspectM

O *framework* AspectM foi implementado de modo a possibilitar a modularização orientada a aspectos dos interesses de mobilidade nos níveis de projeto detalhado e implementação de SMAs móveis. De fato, os objetivos centrais do *framework* AspectM são: (1) promover uma clara separação dos interesses de mobilidade em relação às funcionalidades básicas e aos outros interesses dos agentes, (2) permitir a introdução transparente do código de mobilidade em agentes inicialmente estacionários, e (3) garantir uma integração flexível dos SMAs com várias plataformas de mobilidade.

O *framework* AspectM, além de ter sido construído a partir das diretivas propostas na arquitetura ArchM (Capítulo 5), foi identificado e abstraído a partir da instanciação de diferentes estudos de caso (Capítulo 7) e da abstração de *Aglets* e JADE, duas das principais plataformas de mobilidade existentes (Seção 2.6). O *framework* AspectM foi implementado em *AspectJ* (Kiczales et al., 2001) e pode ser aplicado em vários SMAs móveis.

As seções deste capítulo apresentam a estratégia geral utilizada para a concepção do projeto do *framework*, seus pontos de flexibilização[22], alguns detalhes de projeto e de implementação e o processo de instanciação de uma aplicação a partir das classes e aspectos do AspectM.

6.1.
Funcionalidades Tratadas

A Figura 3 (Seção 2.2), denominada "Ciclo de Vida de Agentes Móveis", ilustra que a existência de um agente móvel pode ser representada segundo as etapas de um modelo que inclui as operações dos protocolos de instanciação, inicialização, movimentação e destruição de agentes. Nesta seção, apresentamos o

22 Do inglês: *hotspots*.

conjunto de funcionalidades tratadas por AspectM. Estas funcionalidades foram detectadas a partir da análise dos protocolos de agentes móveis.

6.1.1. Protocolo de Instanciação

A operação de *instanciação* é executada somente uma vez durante todo o ciclo de vida de um agente. Após sua instanciação, o agente recebe um identificador, tem seu estado interno iniciado e é preparado para instruções adicionais (Seção 2.2). Alguns procedimentos são recorrentes na etapa de instanciação de um agente móvel. Estes procedimentos compõem o protocolo de instanciação de agentes e podem ser descritos através dos seguintes subpassos:

- obter do usuário o identificador do agente móvel na aplicação;
- instanciar o agente móvel na plataforma;
- obter do usuário o tipo e a lista de identificadores de contextos de execução, os quais são utilizados para a criação do itinerário do agente;
- instanciar o itinerário do agente móvel a partir dos dados anteriores;
- inicializar o itinerário a partir de estratégia definida pelo usuário;
- obter do usuário a lista de tarefas que deverão ser movidas junto com o agente móvel durante o seu ciclo de vida;
- obter do usuário a lista de mestres do agente (Seção 3.3), os quais possuem direitos sobre o ciclo de vida do agente móvel;
- incluir na tabela de referências da aplicação uma referência para o agente instanciado na plataforma, utilizando como elemento-chave na tabela seu identificador na aplicação;
- executar operações específicas da aplicação.

6.1.2. Protocolo de Movimentação

As circunstâncias nas quais os agentes devem se movimentar, ou ainda, os instantes em que a ação de movimentação deve ser disparada, são especificados pelos desenvolvedores das aplicações e são denominados pontos de decisão do movimento (Seção 2.2). Os procedimentos do protocolo de movimentação podem ser descritos através dos seguintes subpassos:

- invocar procedimento de verificação da necessidade de movimentação;
- se o teste de verificação retornar um valor favorável, finalizar processos e enviar mensagem de partida para outros agentes;
- efetuar a mudança de contexto de execução do agente móvel, através da invocação de métodos da plataforma.

6.1.3. Protocolo de Inicialização

A *inicialização* é a operação efetuada a cada vez que um agente chega a um novo *host*. A inicialização só é possível porque um agente possui sua própria *thread* de execução (Seção 2.2). Os procedimentos mais comuns no protocolo de inicialização de agentes podem ser descritos através dos seguintes subpassos:

- obter do servidor de agentes o identificador do novo contexto;
- efetuar a configuração do itinerário do agente móvel a partir de dados do novo contexto de execução;
- obter do servidor de agentes o identificador para troca de mensagens do agente móvel em seu novo contexto de execução;
- atualizar na tabela de referências da aplicação os dados do agente móvel em seu novo contexto, como localização e identificador para troca de mensagens, tornando disponível para os outros agentes uma referência atualizada para o agente móvel;
- enviar mensagem de chegada para outros agentes;
- reiniciar ou abrir processos;
- executar operações específicas da aplicação.

6.1.4. Protocolo de Destruição

A execução de uma operação de *destruição* implica na finalização das atividades de um agente, liberando os recursos que são utilizados por ele. Após a destruição, o estado do agente é perdido definitivamente (Seção 2.2). O protocolo de destruição de agentes é descrito através dos seguintes subpassos:

- finalizar todos os processos;
- enviar mensagem notificando a destruição para outros agentes;

- remover da tabela de referências o agente móvel a ser destruído;
- efetivar a destruição do agente móvel, através da invocação de métodos da plataforma.

A Figura 21 resume os subpassos do ciclo de vida dos agentes móveis.

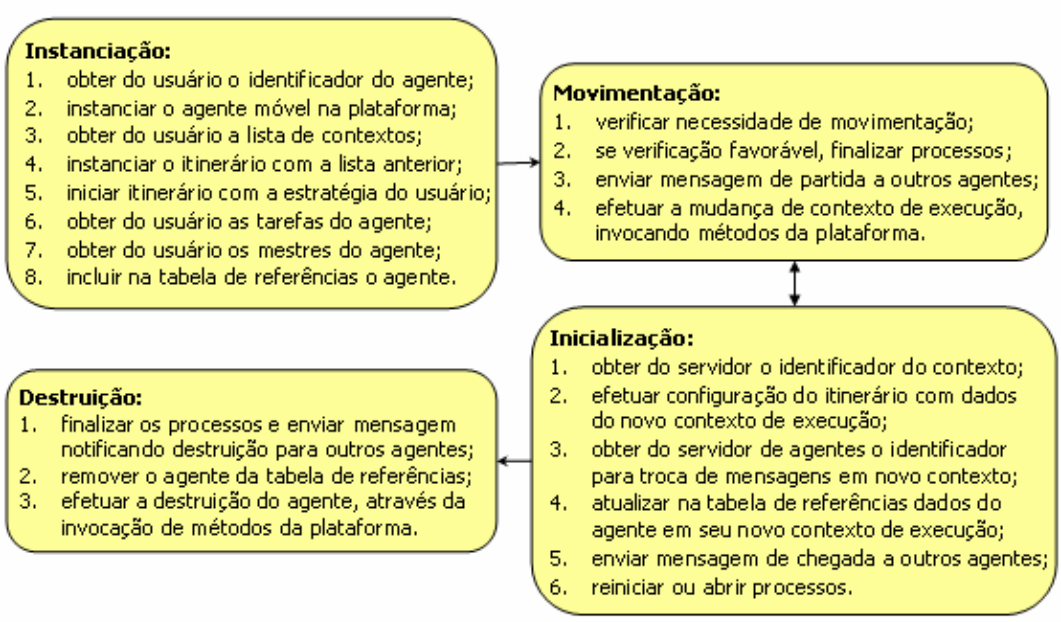

Figura 21. Subpassos do Ciclo de Vida de Agentes Móveis

6.2. Estrutura Geral do Projeto

Inicialmente, supomos a utilização das diretivas impostas pela arquitetura ArchM para modularização dos interesses de mobilidade e fazemos uso somente de abstrações e mecanismos OO. Alcançamos um bom grau em separação de interesses de mobilidade, quando implementamos SMAs móveis seguindo estrutura OO similar à ilustrada na Figura 22. As classes da estrutura OO foram mapeadas a partir dos seguintes componentes de ArchM:

- componente `MobilityPlatform`, consiste nas classes que fazem referências a plataformas de mobilidade específicas (classe `PlatformMobileAgent`, na Figura 22);
- componente `MobilityManagement`, consiste nas classes que possibilitam a flexibilização do uso de plataformas de mobilidade (classes `MobileAgent` e `MobileElement`, na Figura 22);

- componente `MobilityProtocol`, formado por classes que efetuam a execução do protocolo de mobilidade de agentes (classes `MobileElement` e `UserAgent`, na Figura 22); e
- o componente `Kernel`, que consiste nas classes com funcionalidades básicas de agentes da aplicação (classe `UserAgent`, na Figura 22).

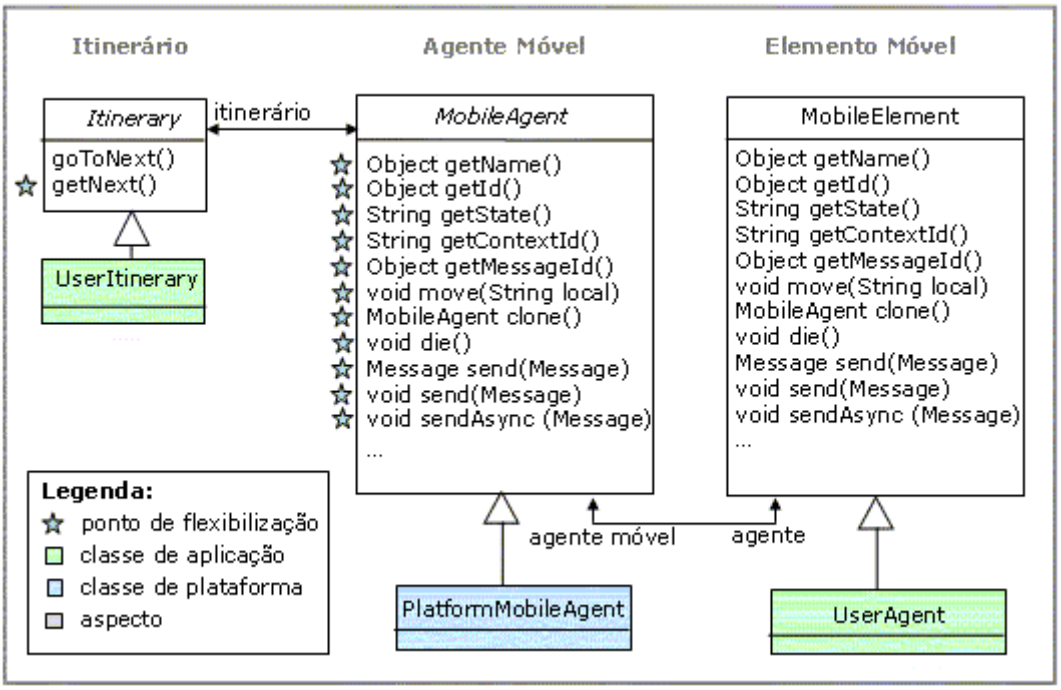

Figura 22. Estrutura OO para Separação de Mobilidade em SMAs

Na Figura 22, a classe abstrata `MobileAgent` constitui o elemento central da estrutura OO e possui como função principal delegar a uma plataforma de mobilidade específica a execução de serviços fornecidos pela classe `MobileAgent`. Constitui, portanto, a realização da interface `IMobileAgent` de ArchM. Como exemplo de métodos da classe `MobileAgent`, têm-se: (i) métodos de identificação do agente e de seu contexto de execução, como `getName()`, `getId()`, `getContextId()` e `getMessageId()`; (ii) métodos que efetuam as operações do ciclo de vida de agentes móveis, como `move()`, `clone()` e `die()`; e, ainda, (iii) métodos para troca de mensagens, como `send()` e `sendAsync()`.

A classe abstrata `Itinerary` é utilizada pela estrutura OO para modularização dos interesses de mobilidade referentes ao itinerário de um agente. Constitui um elemento auxiliar na interface `IMobileAgent` e possui a função de implementar o padrão *Itinerary* (Seção 3.3). Como exemplo de método, tem-se o

método `getNext()`, cuja implementação possui como objetivo a seleção do próximo *host* a ser visitado e depende de estratégia definida pelo desenvolvedor.

Na Figura 22, a classe `MobileElement` consiste no elemento de ligação entre a classe abstrata `MobileAgent` e as classes de tipos ou papéis de agentes móveis em uma aplicação. Por esta razão, seus métodos apenas delegam os serviços invocados nas classes de aplicação para a classe abstrata `MobileAgent`. Adicionalmente, a classe `MobileElement` é utilizada para a especificação do protocolo de mobilidade de tipos ou papéis móveis.

Evidentemente, existem outras classes abstratas necessárias ao alcance da flexibilização de plataformas em SMAs. Para efeitos de simplificação, estas classes não são ilustradas na Figura 22. Nosso objetivo nesta seção é apenas apresentar a estrutura geral de projeto que definimos para a separação explícita de interesses de mobilidade em aplicações. A Seção 6.4 apresenta o projeto detalhado de todas as classes do *framework* AspectM utilizadas para o objetivo de flexibilizar o uso de plataformas em SMAs móveis.

Apesar de o esquema apresentado na Figura 22 permitir uma integração flexível dos SMAs com plataformas de mobilidade, a estrutura OO não promove uma clara separação dos interesses de mobilidade em relação às funcionalidades básicas e outros interesses de agentes. A estrutura OO também não permite a introdução transparente do código de mobilidade em agentes inicialmente estacionários. Estes objetivos não são alcançados, porque persiste a utilização direta de APIs de plataformas de mobilidade, mesmo não se fazendo referência a uma plataforma específica na aplicação (Seção 3.2).

Por exemplo, na Figura 22, a classe `UserAgent` poderá utilizar-se indistintamente de plataformas de mobilidade como *Aglets* e JADE, mas ainda com chamadas diretas a métodos abstraídos a partir das APIs destas plataformas. Por outro lado, como a aplicação deverá implementar métodos abstratos referentes ao ciclo de vida de agentes móveis, obtém-se um projeto com alto grau de acoplamento entre o código da aplicação e o projeto de SMAs móveis induzidos pelo uso destas plataformas, em geral baseado em observadores de eventos[23]. Por esta razão, torna-se difícil a tarefa de fazer com que interesses de mobilidade possam ser introduzidos de maneira transparente nas aplicações.

23 Do inglês: *event listeners*.

O projeto da Figura 22 pode ser melhorado utilizando-se aspectos (Capítulo 4). A utilização de aspectos possibilita o alcance da modularização dos interesses de mobilidade e da introdução transparente de mobilidade em agentes estacionários. O problema de invocar os métodos de mobilidade explicitamente é resolvido pela utilização do próprio conceito de aspectos. Como um aspecto é uma abstração que efetua um entrecorte transversal nos elementos que afeta, é possível fazer com que as classes da aplicação não invoquem os métodos de mobilidade no código de suas funcionalidades básicas. Para isso, é necessário apenas que os pontos de decisão de movimento de um agente estejam bem definidos em seu projeto. Em outras palavras, invertemos o modo como são implementados os interesses de mobilidade em SMAs: abstrações e mecanismos OO, como herança e delegação de métodos, dão lugar a abstrações e mecanismos OA. Tais mecanismos são utilizados de modo a entrecortar pontos de junção do código de funcionalidades básicas e outros interesses de agentes, ao mesmo tempo em que se mantém o encapsulamento dos interesses de mobilidade.

Outro problema da estrutura OO apresentada na Figura 22 diz respeito à definição da hierarquia de classes de um sistema. Não é possível especificar relacionamentos entre classes de maneira a dar livre curso ao usuário na definição da hierarquia de herança entre os agentes de sua aplicação (Seção 3.2). Para resolver este problema, idiomas de linguagens de programação OA possibilitam a extensão de interfaces que não só declaram assinaturas de métodos, mas mantêm funcionamento semelhante ao de classes abstratas. Em outras palavras, é possível utilizar interfaces como se estivéssemos efetivamente utilizando classes abstratas. Desta forma, operações recorrentes são adicionadas ao comportamento de agentes sem impactar o projeto de funcionalidades básicas ou interesses adicionais.

Portanto, utilizando abstrações e mecanismos OA, alcançamos não só a flexibilização de plataformas como a real modularização dos interesses de mobilidade. Na Figura 23, acrescentamos à estrutura OO da Figura 22 o aspecto abstrato `Mobility` e modificamos a definição da classe `MobileElement`. `MobileElement` é reimplementada como uma interface com comportamento semelhante ao de uma classe abstrata. Utilizamos o idioma *AspectJ* denominado *Container Introduction* (Hanenberg et al., 2003), que define a implementação de um tipo específico e especifica quais classes implementam este tipo.

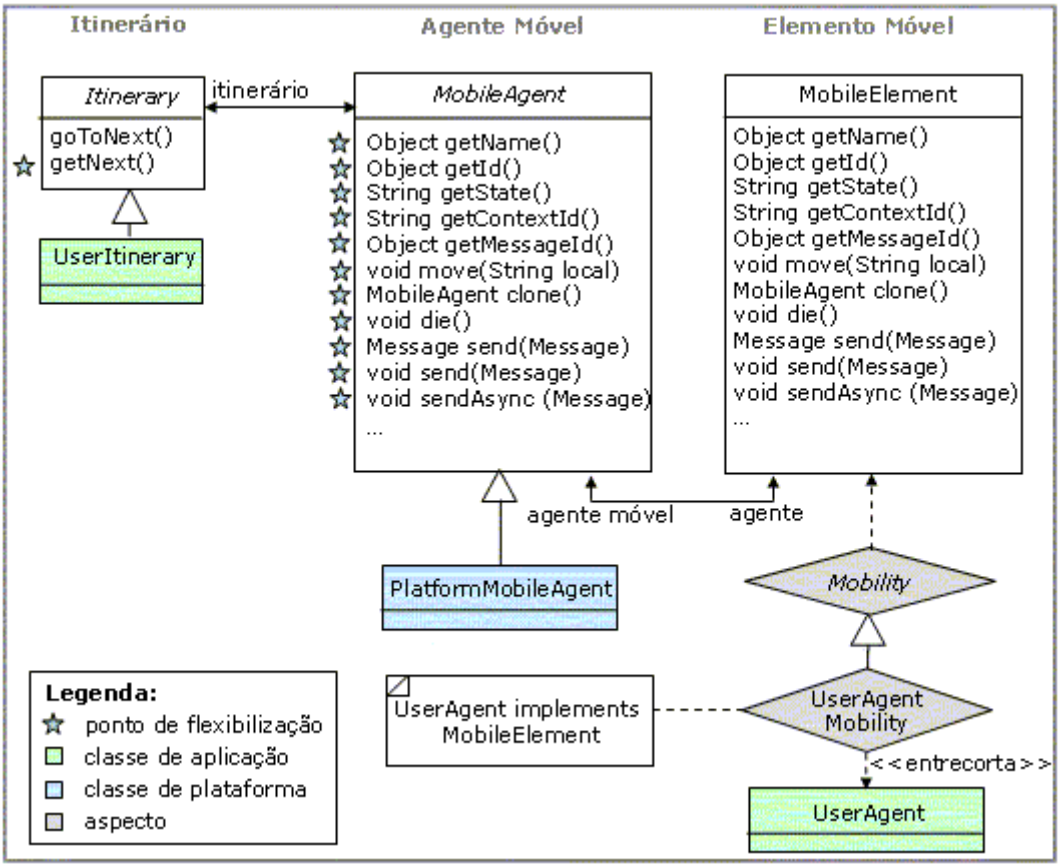

Figura 23. Estrutura OA para Separação de Mobilidade em SMAs

Para utilizar a estrutura OA da Figura 23, seguimos os seguintes passos:

1. para cada classe de agente com requisitos de mobilidade (a classe `UserAgent`, na Figura 23), criamos um aspecto que estende o aspecto abstrato `Mobility` (o subaspecto `UserAgentMobility`, na Figura 23);
2. declaramos a implementação da interface `MobileElement` no aspecto criado no passo anterior (a declaração "`UserAgent implements MobileElement`" em `UserAgentMobility`, na Figura 23);
3. definimos a implementação de métodos e conjuntos de junção abstratos do aspecto `Mobility` no subaspecto `UserAgentMobility`. Os métodos abstratos consistem em procedimentos invocados na execução do protocolo de mobilidade de agentes (Seção 2.2). Os conjuntos de junção são especificados para o disparo da execução dos protocolos de instanciação e de movimentação de agentes.

A Figura 24 ilustra a reimplementação de `MobileElement` para manter funcionamento semelhante ao de classes abstratas.

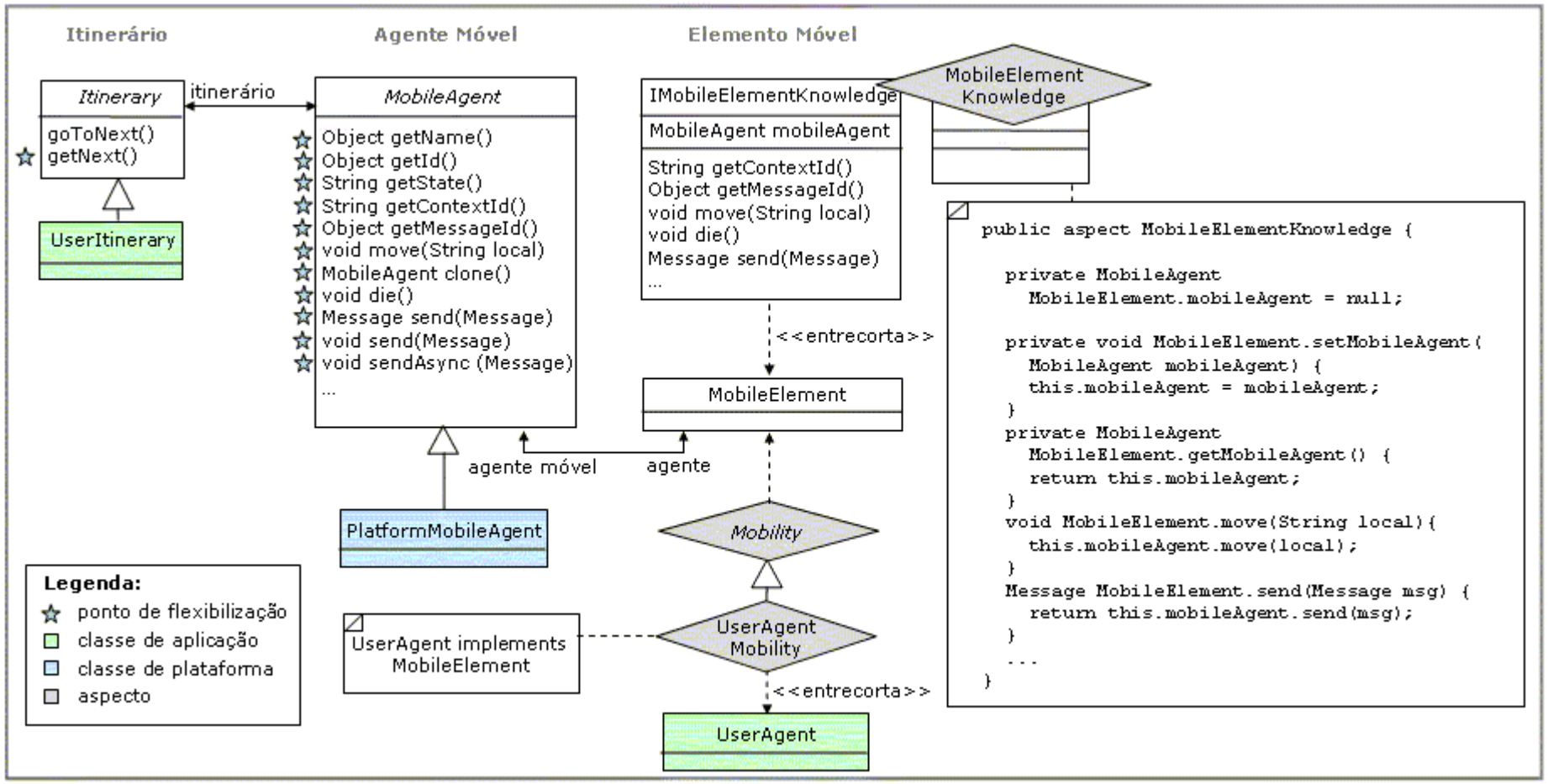

Figura 24. Aplicação de Container Introduction em MobileElement

A Figura 25 tem a finalidade de ilustrar que não é necessário especificar o conjunto de junção de inicialização de um agente no subaspecto de `Mobility`, uma vez que é possível detectá-lo com abstrações e mecanismos OA diretamente do código de APIs de plataformas. Note a introdução dos aspectos `AbstractEvent` e `PlatformEvent` no projeto da Figura 23.

O aspecto `AbstractEvent` é entrecortado pelo código do aspecto `Mobility` quando da execução do método `onArrival()`. Este método é invocado no adendo posterior `onArrival()` do aspecto `AbstractEvent`. Por sua vez, o adendo `onArrival()` é executado após a detecção do conjunto de junção de inicialização, que deve ser concretizado pelo usuário no aspecto `PlatformEvent`.

Na Figura 25, note que o corpo do método `onArrival()` do aspecto `PlatformEvent` não possui linhas de código, uma vez que o objetivo de sua criação foi apenas estabelecer a ponte entre a detecção do método `onArrival()` na plataforma e a detecção do conjunto de junção de inicialização no aspecto `Mobility` do *framework* AspectM.

A estratégia de detecção utilizada para a execução do protocolo de inicialização também aplicada para a execução do protocolo de destruição de agentes móveis. Para efeitos de simplificação, a detecção do conjunto de junção de destruição não é ilustrada na Figura 25. Note que esta mesma estratégia pode ser usada para tornar acessível ao usuário outros eventos no ciclo de vida dos agentes, como pontos de junção importantes na execução de operações como clonagem e troca de mensagens. Embora isto possa ser efetuado com mecanismos de reflexão OO, as abstrações e mecanismos OA permitem detectar estes pontos de junção mais facilmente por sua própria natureza transversal.

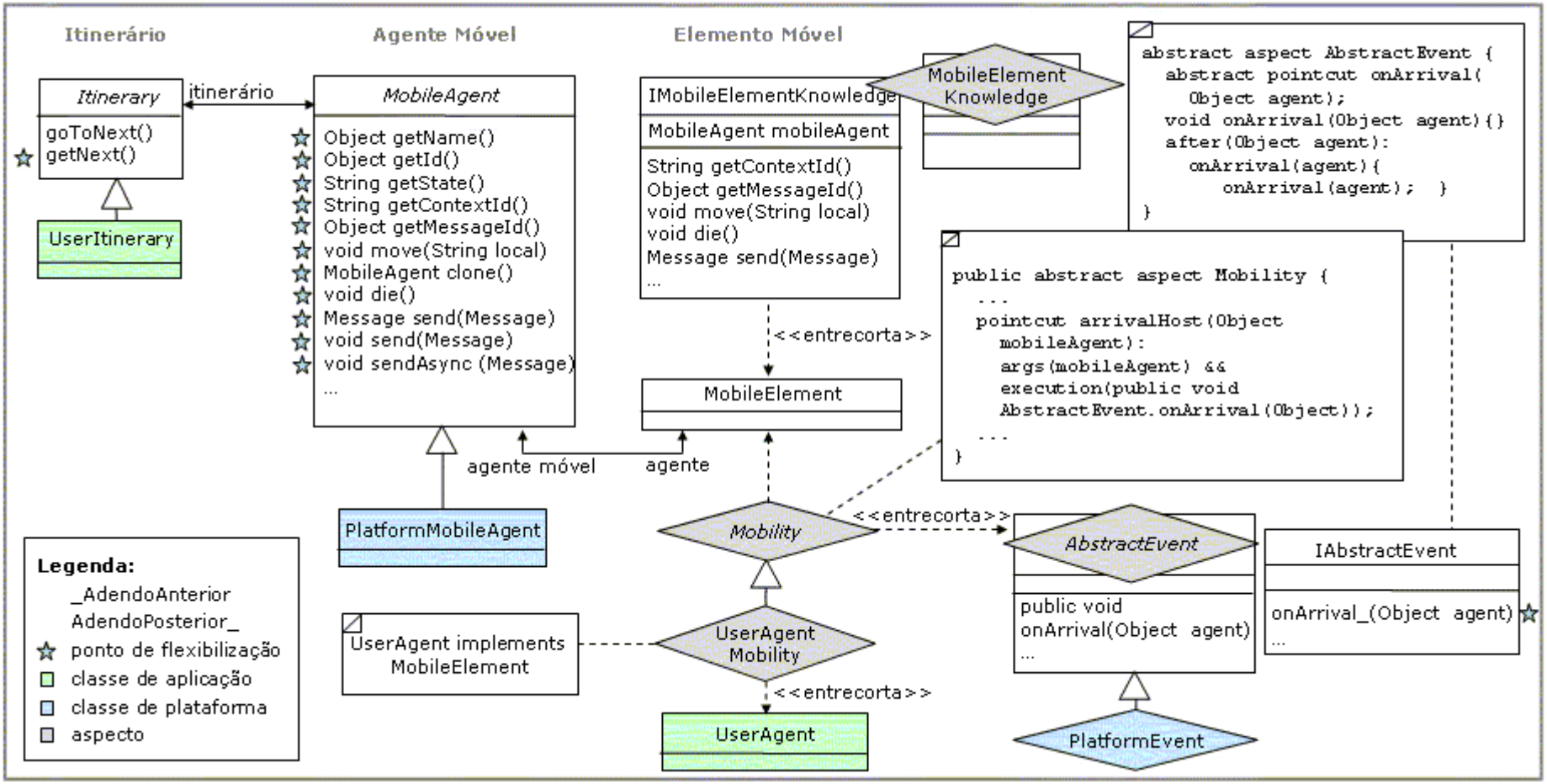

Figura 25. Detecção do Conjunto de Junção de Inicialização

Finalmente, os métodos abstratos do aspecto Mobility consistem em procedimentos invocados na execução do protocolo de mobilidade de agentes (Seção 2.2). O protocolo de mobilidade é implementado pelos adendos no aspecto Mobility. Estes adendos são executados após a detecção de conjuntos de junção do aspecto Mobility que disparam as etapas de instanciação, inicialização, movimentação e destruição de agentes. Os conjuntos de junção de instanciação e de movimentação são abstratos, pois são dependentes de aplicações. Ao contrário, os conjuntos de junção de inicialização e destruição de agentes não são dependentes de aplicações, uma vez que é possível detectá-los com abstrações e mecanismos OA diretamente do código de APIs de plataformas (Figura 25).

Em resumo, a partir da estrutura OA da Figura 23, podemos especificar o mapeamento de ArchM para o *framework* AspectM da seguinte forma:

- o aspecto abstrato `AbstractEvent` constitui a realização da interface `IEvent` de ArchM;
- a classe abstrata `MobileAgent` constitui a realização da interface `IMobileAgent` de ArchM;
- a classe abstrata `Itinerary` e sua subclasse `UserItinerary` constituem elementos auxiliares na realização da interface `IMobileAgent`;
- os serviços da classe `PlatformMobileAgent` constituem a realização da interface `IMobilityServices` de ArchM;
- a declaração intertipo de implementação da interface `MobileElement` no subaspecto de `Mobility` constitui a realização da interface `IMobileElement` de ArchM;
- os adendos de instanciação, inicialização, movimentação e destruição de agentes no aspecto `Mobility` constituem a realização das interfaces `IInstantiation`, `IInitialization`, `IMovement` e `IDestruction` de ArchM, respectivamente;
- os serviços da classe `UserAgent` constituem a realização da interface `IServices` de ArchM.

Da Figura 25, concluímos que: (i) o componente `Kernel` da arquitetura ArchM corresponde à classe `UserAgent`; (ii) o componente `MobilityPlatform` corresponde à classe `PlatformMobileAgent`; (iii) o componente aspectual

`MobilityManagement` corresponde à classe `MobileAgent` e ao aspecto `AbstractEvent`; o componente aspectual `MobilityProtocol` corresponde à interface `MobileElement` e aos aspectos `Mobility` e `UserAgentMobility`.

Comparando à estrutura OO apresentada na Figura 22, temos que: (1) a interface `MobileElement` não é mais parte do componente `MobilityManagement`, passando a pertencer somente ao componente `MobilityProtocol`; (2) a classe `UserAgent` não é mais parte do componente `MobilityProtocol`, passando a pertencer somente ao componente `Kernel`. Portanto, obtemos um projeto que possibilita uma real modularização dos interesses de mobilidade (a classe `UserAgent` está isolada, na Figura 25) e a flexibilização do uso de plataformas de mobilidade (a interface `MobileElement` é o elemento de ligação entre a classe `UserAgent` e a estrutura OA do *framework* AspectM e pertence somente ao componente `MobilityProtocol`, na Figura 25).

Finalmente, a estrutura geral de projeto do *framework* AspectM, apresentada nas Figuras de 23 a 25, também possibilita o alcance da introdução transparente de mobilidade em agentes inicialmente estacionários. A introdução dos interesses de mobilidade corresponde à tarefa de concretização do aspecto abstrato `Mobility` para um agente estacionário. A retirada da mobilidade consiste em remover este aspecto do projeto do agente.

Evidentemente, se o projeto da aplicação não utilizar diligentemente as abstrações e os mecanismos OA, o resultado da tarefa de introduzir ou retirar o aspecto concreto de `Mobility` poderá ter impacto no funcionamento geral do sistema. Isto ocorre na situação em que ainda persiste alto grau de acoplamento entre funcionalidades básicas e interesses de mobilidade mesmo utilizando-se aspectos no projeto do agente. Isto é conseqüência do projeto da aplicação e não da incapacidade da OA em modularizar os interesses de mobilidade.

6.3. Projeto Detalhado do Componente *MobilityProtocol*

O projeto detalhado do *framework* AspectM segue as diretivas impostas pela arquitetura ArchM (Capítulo 5). Portanto, o componente `Kernel` e os demais componentes arquiteturais, com exceção dos componentes `MobilityManagement` e `MobilityProtocol`, são implementados como um conjunto de classes. Os

componentes `MobilityManagement` e `MobilityProtocol` são refinados como um conjunto de aspectos e classes auxiliares. Nestes componentes, aspectos são usados para isolar a implementação dos interesses de mobilidade das classes que implementam os outros interesses de um agente. O foco desta seção é a descrição do projeto detalhado do componente `MobilityProtocol`. A Seção 6.4 descreve o projeto detalhado do componente `MobilityManagement`.

6.3.1. Estrutura do Componente MobilityProtocol

A Figura 26 ilustra o projeto de uma aplicação instanciada a partir do *framework* AspectM, com detalhamento do componente `MobilityProtocol` através de elementos gráficos da linguagem ASideML (Seção 4.3). O componente `MobilityProtocol` é composto pelo aspecto abstrato `Mobility` e pelos aspectos que tornam concreto o aspecto `Mobility` (o aspecto `UserAgentMobility`, na Figura 26). A interface `IMobileElement` do componente `MobilityProtocol` é realizada por declarações intertipos relativas à implementação das interfaces `MobileElement` (a declaração "`UserAgent implements MobileElement`", na Figura 26) e `Serializable` (a declaração "`UserTask implements Serializable`", na Figura 26) nos subaspectos de `Mobility` (em `UserAgentMobility`, na Figura 26).

Os adendos do aspecto `Mobility` constituem a realização das interfaces `IInstantiation`, `IInitialization`, `IMovement` e `IDestruction` do componente `MobilityProtocol`. Os conjuntos de junção dos adendos de instanciação e movimentação são abstratos no aspecto `Mobility` (Seção 6.2). Os subaspectos de `Mobility` devem especificar estes conjuntos de junção para obter os serviços de mobilidade de uma plataforma (conjuntos de junção `agentInstantiation()` e `agentMovement()` em `UserAgentMobility`, na Figura 26). Além das interfaces transversais implementadas por declarações intertipos, conjuntos de junção e adendos associados, o aspecto `Mobility` possui métodos internos. Estes métodos são abstratos no aspecto `Mobility`, pois são dependentes de aplicação. Alguns exemplos de métodos internos do aspecto `Mobility` são: `getItineraryList()`, `getMasterList()` e `doAfterAgentInstantiation()` (estes métodos são especificados em `UserAgentMobility`, na Figura 26).

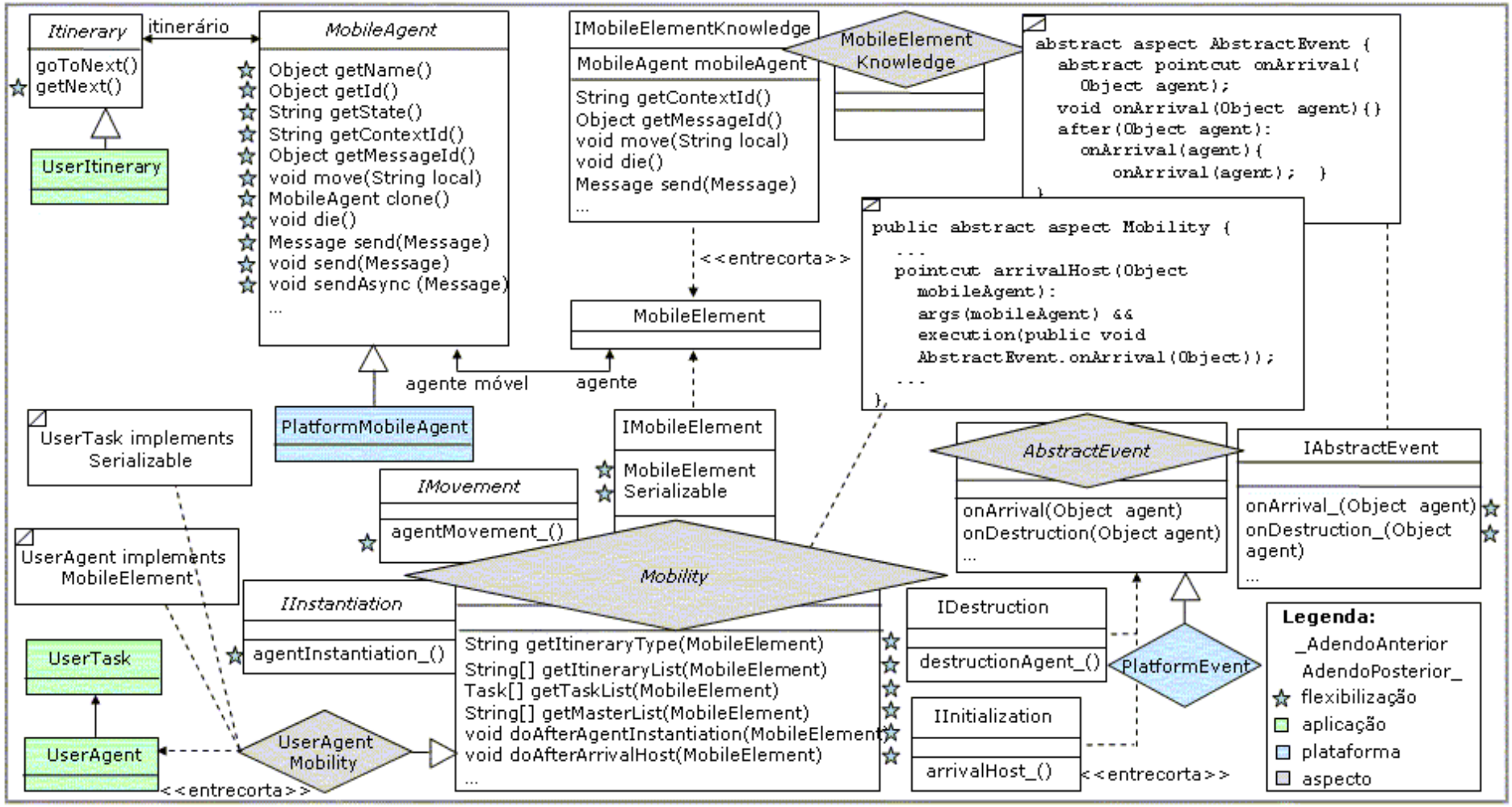

Figura 26. Projeto Detalhado do Componente MobilityProtocol

A estrutura do componente `MobilityProtocol` mantém referências a classes do componente `Kernel`. Estas classes modularizam as funcionalidades básicas e o conhecimento intrínseco de agentes em SMAs. O componente `Kernel` também implementa a interface `IServices`, que define os serviços fornecidos por um agente a seus clientes (Seção 6.2). As classes do componente `Kernel` não pertencem ao *framework* AspectM; podem, entretanto, ser subclasses de classes pertencentes ao *framework*. Na Figura 26, as classes do componente `Kernel` são representadas pelas classes `UserAgent`, `UserTask` e `UserItinerary`.

6.3.2. Dinâmica do Componente MobilityProtocol

Tendo em vista a estrutura OA do *framework* AspectM (Seção 6.2) e o projeto detalhado do componente `MobilityProtocol` (Seção 6.3.1), o protocolo de instanciação do ciclo de vida de agentes móveis (Figura 21) pode ser ilustrado seguindo o esquema de delegação ilustrado na Figura 27.

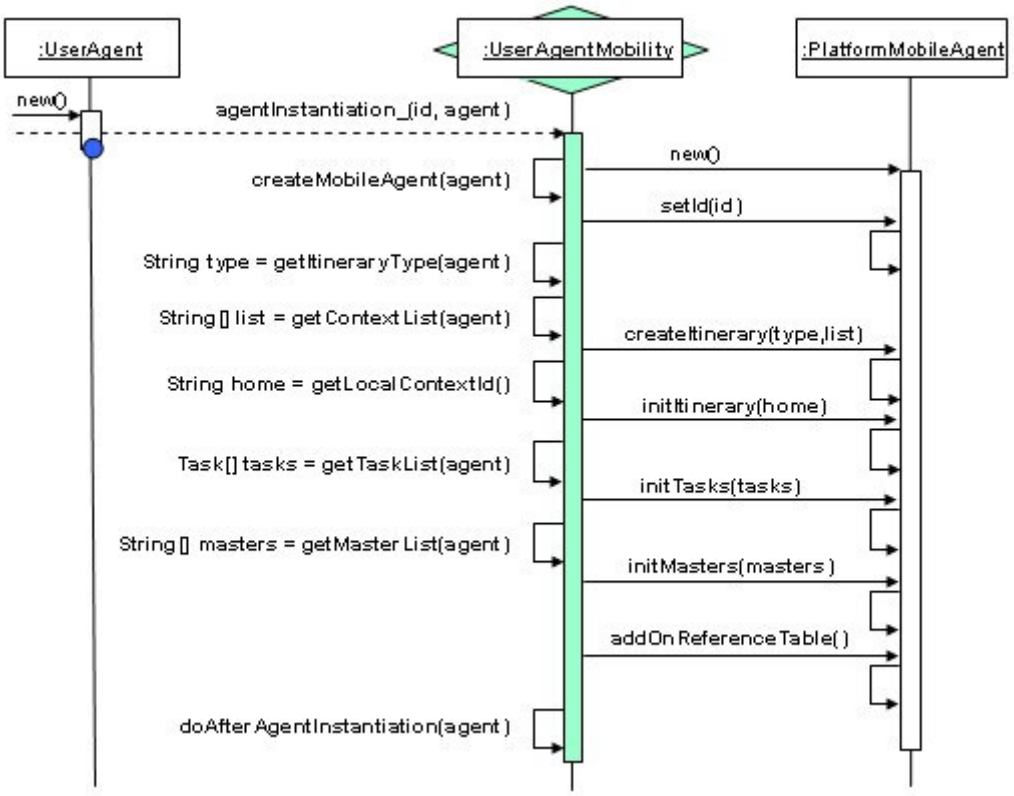

Figura 27. Protocolo de Instanciação no Componente MobilityProtocol

Na Figura 27, o agente `UserAgent` executa os seguintes passos:

- obtém do usuário o identificador do agente móvel a ser instanciado na plataforma de mobilidade (parâmetro `id` do conjunto de junção `agentInstantiation` no aspecto `UserAgentMobility`);
- obtém do usuário uma referência para o agente que requer serviços de mobilidade na aplicação (parâmetro `agent` do conjunto de junção `agentInstantiation` no aspecto `UserAgentMobility`);
- instancia um agente móvel na plataforma para o agente declarado como elemento móvel na aplicação (método `createMobileAgent()`);
- configura o identificador do agente móvel (método `setId()`);
- obtém do usuário o tipo (método `getItineraryType()`), a lista de contextos de execução (método `getContextList()`) e outros dados importantes para a instanciação do itinerário do agente;
- instancia o itinerário do agente móvel a partir dos dados anteriores (método `createItinerary()`);
- obtém do agente móvel instanciado na plataforma o identificador do contexto de execução do agente (método `getLocalContextId()`);
- inicializa o itinerário com o identificador do contexto e a partir de estratégia indicada pelo usuário (método `initItinerary()`);
- obtém do usuário a lista de tarefas que deverão ser movidas junto com o agente móvel durante o seu ciclo de vida (método `getTaskList()`);
- associa as tarefas com o agente móvel (método `initTasks()`);
- obtém do usuário a lista de mestres do agente, os quais possuem plenos direitos sobre o ciclo de vida do agente móvel instanciado na plataforma (método `getMasters()`);
- associa a lista de mestres com o agente móvel instanciado na plataforma (método `initMasters()`);
- inclui no conjunto de referências da aplicação o agente instanciado, utilizando-se como elemento-chave na tabela o identificador do agente móvel na aplicação (método `addOnReferenceTable()`);
- executa os procedimentos especificados para serem executados na aplicação após a instanciação do agente (no método `doAfterAgentInstantiation()`).

O protocolo de movimentação de agentes é ilustrado na Figura 28.

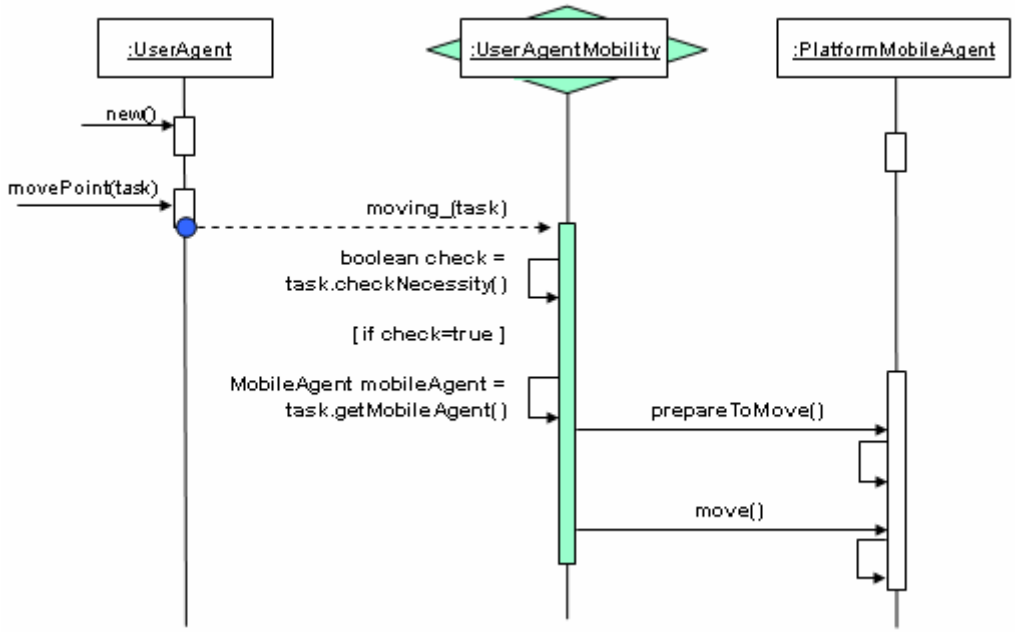

Figura 28. Protocolo de Movimentação no Componente MobilityProtocol

Na Figura 28, o agente que está na iminência de migrar para outro *host*:

- obtém do usuário dados sobre o ponto de decisão de movimento (parâmetro `task`);
- com os dados obtidos no passo anterior, invoca o procedimento que verifica a necessidade de movimentação (método `checkNecessity()`);
- se o teste de verificação retornar um valor favorável, executa os procedimentos de partida do agente (método `prepareToMove()`);
- efetua a mudança de contexto de execução do agente móvel, pela invocação de métodos da plataforma de mobilidade (método `move()`).

Um detalhe sutil no projeto do componente `MobilityProtocol` é a definição da interface `Task`. Esta interface é utilizada em duas situações: (1) para definir um tipo comum para classes a partir das quais são instanciados objetos que compõem o conhecimento intrínseco de um agente, como no caso dos objetos retornados pelo método `getTaskList()` do protocolo de instanciação (Figura 27), ou (2) para definir uma interface comum para objetos alvos[24] utilizados na definição de conjunto de junção do protocolo de movimentação (Figura 28).

24 Do inglês: *targets*.

O protocolo de inicialização de agentes móveis é ilustrado na Figura 29.

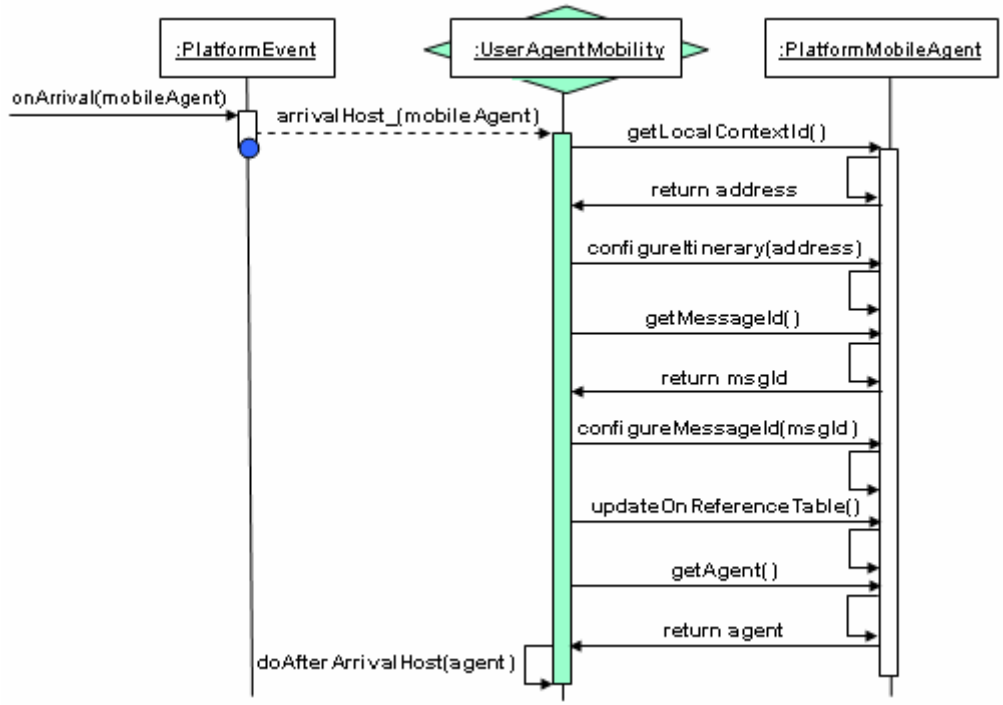

Figura 29. Protocolo de Inicialização no Componente MobilityProtocol

Na Figura 29, o agente que chega a um novo *host*:

- obtém do servidor de agentes da plataforma o identificador de novo contexto de execução (método `getLocalContextId()`);
- efetua a configuração do itinerário do agente a partir de dados do novo contexto de execução (método `configureItinerary()`);
- obtém do servidor plataforma o identificador para troca de mensagens com o agente móvel no novo contexto (método `getMessageId()`);
- efetua a configuração de atributos referentes à troca de mensagem a partir do identificador obtido na linha anterior (método `configureMessageId()`);
- atualiza a localização e o identificador para troca de mensagens da referência ao agente móvel armazenada no conjunto de referências da aplicação (método `updateOnReferenceTable()`);
- executa os procedimentos especificados para serem executados na aplicação após a chegada do agente a um novo *host* (no método `doAfterArrivalHost()`).

O protocolo de destruição de agentes é ilustrado na Figura 30.

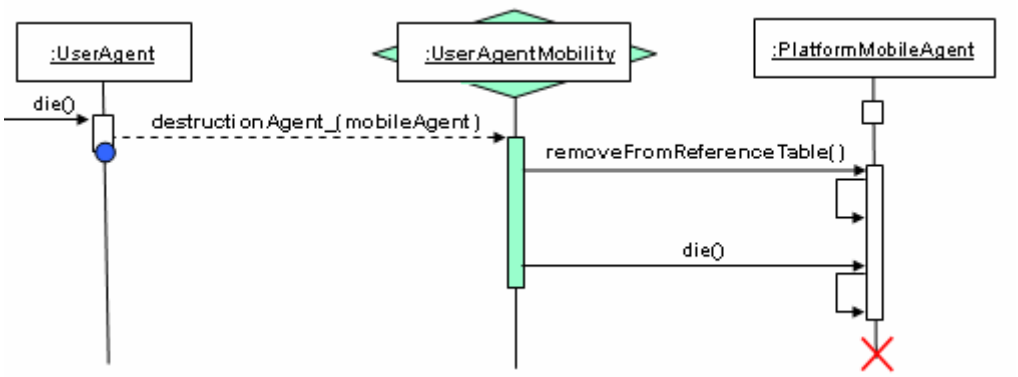

Figura 30. Protocolo de Destruição no Componente MobilityProtocol

Na Figura 30, o agente a ser destruído na plataforma:

- remove do conjunto de referências da aplicação a referência para o agente sendo destruído (método `removeFromReferenceTable()`);
- efetua a destruição do agente móvel, pela invocação de métodos da plataforma de mobilidade (método `die()`).

Nas Figuras 27 e 29, note que os métodos `doAfterAgentInstantiation()` e `doAfterArrivalHost()` são abstratos no aspecto `Mobility`. A concretização deste método no subaspecto `UserAgentMobility` corresponde à tomada do fluxo de execução para os métodos específicos da aplicação. Observe nas Figuras 28 e 30 que isto não ocorre nas etapas de movimentação e destruição, onde o fluxo de execução não é retornado ao usuário.

6.3.3. Processo de Instanciação de Aplicação

Um dos objetivos do *framework* AspectM é possibilitar a integração flexível de SMAs com plataformas de mobilidade. Por exemplo, para integrar o *framework* AspectM com *Aglets*, um pacote de classes específico para integração de AspectM com a plataforma *Aglets* deve ser desenvolvido. Como a implementação destas classes independe dos requisitos específicos de aplicações, o pacote de integração do AspectM com a plataforma *Aglets* é reutilizável no processo de instanciação de qualquer aplicação que opte por fazer uso dos serviços de mobilidade providos por esta plataforma.

Evidentemente que, dependendo dos requisitos de uma aplicação específica, este pacote poderá ser modificado a fim de que seja possível fornecer uma solução adequada à tarefa de integração do *framework* AspectM com várias plataformas de mobilidade. As questões relacionadas à flexibilização de plataformas no projeto do *framework* AspectM são apresentadas na Seção 6.4. Nesta seção, supomos que o pacote referente à plataforma escolhida por um usuário já está disponível. Os usuários devem adotar os seguintes passos no processo de instanciação de uma aplicação a partir do *framework* AspectM:

1. para um subaspecto de `Mobility`, é especificado o tipo ou papel de agente que deve ter acesso aos serviços de mobilidade, através de declarações intertipos do tipo "`implements MobileElement`";
2. para um subaspecto de `Mobility`, são especificados os tipos de objetos que compõem a lista de tarefas de um agente ou papel móvel, através de declarações intertipos do tipo "`implements Task`";
3. para um subaspecto de `Mobility`, são especificados os tipos de objetos que podem ser movidos junto com os agentes, através de declarações intertipos do tipo "`implements Serializable`";
4. se necessário, definição de classes de itinerário, estendendo a interface `Itinerary`. Devem ser concretizados métodos como `getNext()`;
5. para cada subaspecto de `Mobility`, devem ser especificados os métodos `getItineraryType()` e `getContextList()`, se a noção de itinerário é utilizada. Assume-se que as classes de itinerário já tenham sido implementadas (passo 4);
6. para um subaspecto de `Mobility`, é especificado o conjunto de junção que dispara o protocolo de instanciação do elemento móvel;
7. para um subaspecto de `Mobility`, são especificados os métodos de aplicação executados no protocolo de instanciação do elemento móvel. Estes métodos devem ser invocados na concretização do método `doAfterAgentInstantiation()` do aspecto `Mobility`;
8. para um subaspecto de `Mobility`, é especificado o conjunto de junção que dispara o protocolo de movimentação do elemento móvel;

9. para um subaspecto de `Mobility`, são especificados métodos do protocolo de movimentação, como `checkNecessity()`, que verifica a necessidade de movimentação de um agente em um contexto específico;
10. para um subaspecto de `Mobility`, são especificados os métodos de aplicação executados no protocolo de inicialização do elemento móvel. Estes métodos devem ser invocados na concretização do método `doAfterArrivalHost()` do aspecto `Mobility`.

Por exemplo, a Figura 31 apresenta o aspecto `UserAgentMobility`, que estende o aspecto abstrato `Mobility` a fim de definir o comportamento de mobilidade específico para a classe `UserAgent`. No aspecto `UserAgentMobility`, são especificados os pontos de flexibilização do *framework* para o contexto de execução do agente `UserAgent`.


```
1: public aspect UserAgentMobility extends Mobility {
2: declare parents: UserAgent implements MobileElement;
3: declare parents: UserTask implements Task;
4: declare parents: UserObject implements Serializable;
5: pointcut agentInstantiation(String referenceName,
6:    MobileElement agent): this(agent) && args(referenceName,*)
7:    && initialization(UserAgent+.new(String,*));
8: void doAfterAgentInstantiation(MobileElement agent){...}
9: String getItineraryType(MobileElement agent){...}
10: String[] getContextList(MobileElement agent){...}
11: Task[] getTaskList(MobileElement agent){...}
12: String[] getMasterList(MobileElement agent){...}
13: void doAfterArrivalHost(MobileElement agent) {...}
14: pointcut agentMovement(Task task):
15:    this(task) && execution(Hashtable
16:    UserTask.executeTask(Vector));
17: ...
18: }
```

Figura 31. Instanciação do AspectM para o Agente UserAgent

Na Figura 31, são especificados no aspecto `UserAgentMobility` os seguintes pontos de flexibilização do *framework* AspectM:

- definição de elementos móveis. O tipo de agente `UserAgent` implementa a interface `MobileElement` (linha 2). Através da utilização da cláusula *declare parents*, todos os atributos e métodos da interface `MobileElement` são herdados pela classe `UserAgent`. O comportamento definido na interface `MobileElement` é adicionado à classe `UserAgent` durante o processo de combinação do sistema (Capítulo 4);
- definição da lista de tarefas do elemento móvel. A classe de tarefa `UserTask` implementa a interface `Task` (linha 3). Através da utilização da cláusula *declare parents*, todos os atributos e métodos da interface `Task` são implicitamente herdados pela classe `UserTask`. No *framework* AspectM, a interface `Task` é utilizada para definir as tarefas que devem fazer parte do conhecimento intrínseco de um agente móvel e/ou tarefas utilizadas na definição de conjunto de junção de movimentação. A inclusão de um objeto `UserTask` na lista de tarefas de um agente `UserAgent` é efetuada no método `getTaskList()`, na linha 11);
- definição de elementos serializáveis, como `UserObject` (linha 4);
- definição do conjunto de junção `agentInstantiation()` (linhas de 5 a 7). Especificação de que o construtor de `UserAgent` é o ponto de junção onde o comportamento de mobilidade deve ser adicionado ao agente `UserAgent` sendo instanciado;
- definição do método `doAfterAgentInstantiation()` (linha 8). Especificação de métodos específicos de aplicação executados após a inicialização do agente na plataforma de mobilidade;
- definição de procedimentos do protocolo de inicialização (linhas 9 a 12), como `getItineraryType()`, `getContextList()`, `getTaskList()` e `getMasterList()`;
- definição de procedimentos que devem ser executados após a chegada em um novo *host* (linha 13). Estes procedimentos são invocados no corpo do método `doAfterArrivalHost()`;

- definição do conjunto de junção `agentMovement()` (linhas de 14 a 16). Especificação de que a execução do método `executeTask()` da classe `UserTask` é um ponto de decisão de movimento.

A Figura 32 ilustra o protocolo de movimentação do agente `UserAgent`.

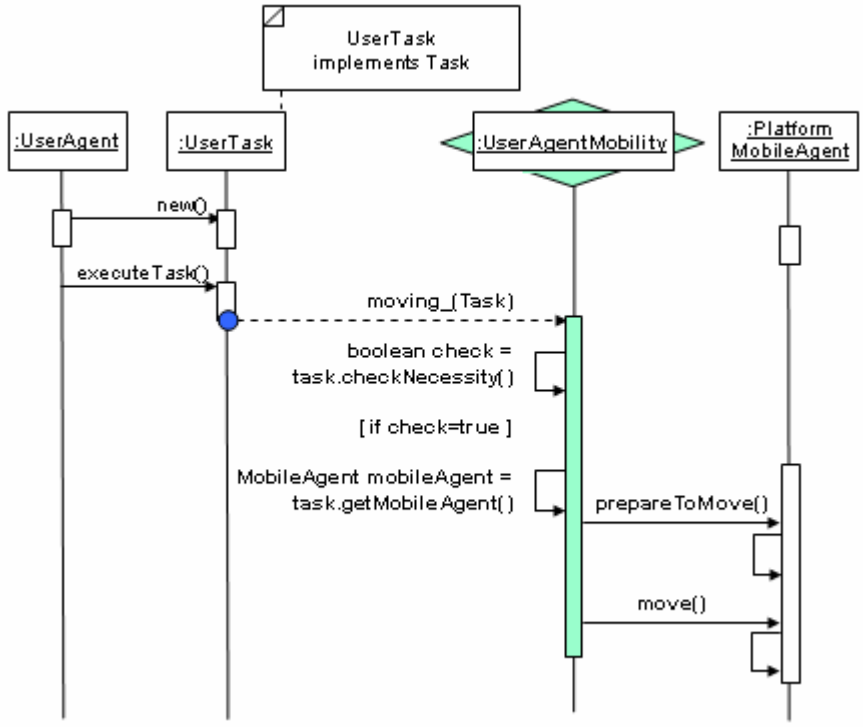

Figura 32. Protocolo de Movimentação do Agente UserAgent

O protocolo de movimentação ilustrado na Figura 32 corresponde à seguinte seqüência de passos na execução da tarefa `UserTask`:

1. instanciação de um objeto `UserTask`, que corresponde à criação de uma tarefa de `UserAgent`. O plano `UserTask` implementa a interface `Task` e, por essa razão, pode ter seu contexto de execução exposto no conjunto de junção `agentMovement()`;
2. o aspecto `UserAgentMobility` detecta a execução do método `executeTask()` na tarefa `UserTask`, através do conjunto de junção `agentMovement()`;
3. o aspecto `UserAgentMobility` captura as informações do contexto de execução da tarefa `UserTask`. Neste caso, a informação relevante é o valor de retorno do método `executeTask()`;

4. `UserAgentMobility` executa o método `checkNecessity()` para verificar a necessidade de se movimentar o agente `UserAgent`. Note que o método `checkNecessity()` é especificado pelo usuário;
5. o aspecto `UserAgentMobility` invoca o método `prepareToMove()`, a fim de que sejam executadas as últimas ações antes da transferência;
6. o aspecto `UserAgentMobility` invoca o método `move()`, que delega à plataforma de mobilidade a tarefa de efetivamente movimentar o agente.

Uma observação importante no contexto de instanciação de agentes ou papéis móveis é que o conjunto de junção de instanciação (`agentInstantiation()`) deve apontar para um tipo de agente mesmo quando o cenário de movimentação é específico de um papel. Isto se dá pelo fato de que, quando um papel de agente se move, o tipo de agente associado deve também ser movido, uma vez que um papel pode manipular o conhecimento contido nas classes que compõem o tipo de agente. A definição do papel *chair* no sistema *Expert Committee* ilustra a instanciação do *framework* AspectM para o contexto de execução de papéis (Seção 7.2)

6.4. Projeto Detalhado do Componente *MobilityManagement*

O projeto e a implementação do *framework* AspectM são orientados pelos seguintes objetivos: uma clara separação dos interesses de mobilidade em relação às funcionalidades básicas e outros interesses dos agentes, uma introdução transparente do código de mobilidade em agentes inicialmente estacionários e uma integração flexível dos SMAs com diferentes plataformas de mobilidade. Como conseqüência do tratamento conjunto destes objetivos, há a necessidade de categorização dos pontos de flexibilização do AspectM em dois grupos. Tais grupos de pontos de flexibilização são derivados da visão que os usuários possuem de AspectM durante o processo de desenvolvimento de aplicações.

O primeiro grupo de pontos de flexibilização engloba aqueles que os desenvolvedores de SMAs móveis utilizam quando os pacotes de comunicação entre o *framework* AspectM e a plataforma de mobilidade já estão desenvolvidos e dizem respeito à instanciação de uma aplicação a partir do AspectM. Estes

pontos de flexibilização foram descritos na Seção 6.3. O segundo grupo de pontos de flexibilização diz respeito àqueles que são utilizados para o desenvolvimento de pacotes de classes que efetuam a comunicação entre o *framework* AspectM e as plataformas de mobilidade.

É possível também que no processo de desenvolvimento de SMAs móveis todos os pontos de flexibilização sejam utilizados em conjunto. Isto ocorre quando ainda não estão implementadas as classes que devem fazer a comunicação entre o AspectM e a plataforma de mobilidade escolhida, ou quando estas classes necessitam de funcionalidades adicionais para atender aos requisitos de SMAs em desenvolvimento. O restante desta seção trata do desenvolvimento de classes que fazem a comunicação entre o AspectM e as plataformas de mobilidades.

6.4.1. Necessidade de Flexibilização de Plataforma

Os adendos de instanciação, inicialização e destruição de agentes móveis do *framework* AspectM seguem a estratégia geral de manter uma tabela de referências para os agentes instanciados em uma aplicação (Seção 6.1). A necessidade de manutenção de uma tabela de referências deve-se a duas razões principais: (1) as plataformas de mobilidade possuem implementações muito específicas para o tratamento do conceito de contexto de execução, e (2) existem diferenças significativas entre os mecanismos de comunicação disponibilizados pelas plataformas de mobilidades.

Por exemplo, JADE permite gerenciamento remoto dos contextos de execução da plataforma, enquanto a plataforma *Aglets* não disponibiliza este serviço. JADE mantém três agentes centrais para a tarefa de administração remota: o Agente de Administração do Sistema, que controla o acesso à plataforma sendo responsável pela autenticação e controle de inscrições de agentes; o Agente de Canal de Comunicação, responsável pela comunicação entre agentes internos ou externos a plataforma; o Diretório Facilitador, que provê um serviço de páginas amarelas para a plataforma (Seção 2.6.2). Na plataforma *Aglets*, estes serviços não estão disponíveis; cabe ao desenvolvedor implementar tarefas como a busca por agentes nos contextos de execução. Para este fim, o programador *Aglets* utiliza as abstrações Contexto e *Proxy* (Seção 2.6.1).

Comparando-se as plataformas JADE e *Aglets*, temos que:

- embora JADE disponibilize serviços como a busca por descritores de agentes em ambientes remotos, o código para a comunicação com os agentes centrais da plataforma é em geral replicado por toda a aplicação. Além disso, uma quantidade considerável de linhas de código é gasta na comunicação com os agentes centrais da plataforma. Em *Aglets*, um *proxy* fornece transparência de localização aos agentes, isto é, a comunicação com o *aglet* é efetuada como se estivesse ocorrendo localmente, porém, é necessário obter o *proxy* do *aglet* correspondendo à sua interface de comunicação em cada *host*;
- em JADE, é possível efetuar a troca de mensagens a partir dos descritores de agentes retornados pelos gerenciadores da plataforma. Dentre outras funções, um descritor retorna o identificador de um agente, que, por sua vez, é usado para a especificação deste agente como um dos receptores de uma mensagem. Em *Aglets*, o receptor de uma mensagem deve ser especificado através do *proxy*, sendo necessário mantê-lo atualizado para que a comunicação seja efetivada.

Portanto, para que seja possível flexibilizar o uso de plataformas em SMAs móveis, é necessário encapsular o serviço de gerenciamento de contexto (como no caso de JADE), ou implementá-lo e mantê-lo encapsulado, para o caso daquelas que não possuem este serviço (no caso de *Aglets*). Evidentemente, desenvolvemos os serviços de gerenciamento de contexto do *framework* AspectM tendo em vista apenas os serviços de gerenciamento mais comuns nos SMAs.

6.4.2. Estrutura do Componente MobilityManagement

As diretivas para composição do componente `MobilityManagement` na arquitetura ArchM (Capítulo 5) praticamente são deixadas em aberto, uma vez a maioria dos elementos que compõem este componente são implementados fazendo uso de abstrações e mecanismos OO.

A arquitetura ArchM apenas especifica a necessidade de definição de duas interfaces: (1) `IMobileAgent`, que permite acesso aos serviços de agentes móveis

freqüentemente utilizados em SMAs e que também estão disponíveis nas diferentes plataformas de mobilidade, e (2) `IEvent`, que entrecorta a chamada ou a execução de serviços de plataformas de mobilidade (Seção 5.3).

Para alcançarmos o objetivo de flexibilização de plataformas e tendo em vista as diretivas de ArchM para o projeto do componente `MobilityManagement`, definimos os seguintes pontos de flexibilização no *framework* AspectM:

- a classe `ReferenceCreator`, que implementa o padrão *Singleton* (Gamma et al., 1995) e possui um método `getInstance()` retornando uma instância da classe para cada contexto onde agentes podem ser instanciados. As responsabilidades do `ReferenceCreator` são: (i) criar um contexto local para a execução de agentes móveis, (ii) instanciar os agentes móveis no contexto de execução anterior, e (iii) tornar acessíveis os serviços de mobilidade para agentes móveis instanciados;
- a classe `ReferenceManager`, que implementa o padrão *Singleton* (Gamma et al., 1995) e com uma única instância para toda a aplicação. Esta instância é obtida através do método `getInstance(Object)`. O argumento deste método deve corresponder ao contexto onde está localizada a tabela de referências da aplicação. O `ReferenceManager` possui as responsabilidades de: (i) instanciar a tabela de referências da aplicação, (ii) atualizar a tabela a cada instanciação, inicialização ou destruição de agentes, (iii) responder a solicitações de buscas por agentes em contextos. O `ReferenceManager` deve ser implementado de modo que esteja sempre acessível remotamente;
- a interface `MobileAgent`, com a responsabilidade de definir a assinatura de serviços usualmente necessários aos desenvolvedores de SMAs móveis, inclusive a troca de mensagens com outros agentes da aplicação. Além disso, `MobileAgent` deve também ser capaz de se comunicar com a instância de `ReferenceManager`, na busca por agentes em contextos ou para a atualização da tabela de referências;
- o aspecto `AbstractEvent`, cuja responsabilidade é possibilitar a detecção de pontos de junção importantes no ciclo de vida de agentes móveis, como inicialização, clonagem e destruição de agentes;

- finalmente, a classe `MessageParser`, com a responsabilidade de efetuar a tradução do formato de mensagens da plataforma de mobilidade para o formato do *framework* AspectM e vice-versa.

As responsabilidades das classes `ReferenceCreator` e `ReferenceManager` e da interface `MobileAgent` são os serviços que normalmente aparecem espalhados e entrelaçados no código de SMAs móveis. As responsabilidades do aspecto `AbstractEvent` e da classe `MessageParser` são necessárias para o propósito de modularização dos interesses de mobilidade do *framework* AspectM. As classes `ReferenceCreator`, `ReferenceManager`, `MobileAgent` e `MessageParser` constituem em conjunto a realização da interface `IMobileAgent` do componente `MobilityManagement`. Estas classes permitem acesso aos serviços de agentes móveis freqüentemente utilizados em SMAs. O aspecto `AbstractEvent` constitui a realização da interface `IEvent`, uma vez que permite entrecortar a execução de serviços importantes de uma plataforma de mobilidade

A Figura 33 apresenta o projeto do *framework* AspectM com foco nos elementos de projeto utilizados para o alcance do objetivo de flexibilização de plataformas. Na Figura 33, `ReferenceCreator` é uma classe abstrata que disponibiliza os seguintes métodos principais em sua interface: (i) `createLocalContext()`, que instancia um contexto local para a execução de agentes; (ii) `createAgent()`, que instancia um representante móvel para um agente que declara a implementação da interface `MobileElement`; (iii) `startAgent()`, que torna acessíveis os serviços de mobilidade para os representantes móveis instanciados na plataforma; (iv) `createMobileAgent()`, um *Template Method* (Gamma et al., 1995) que invoca os três métodos anteriores (Figura 33). O *template method* de criação de agentes é invocado durante a execução do protocolo de instanciação de agentes móveis (Seção 6.4.3). No AspectM, as classes `ReferenceManager` e `MobileAgent` são implementadas como interfaces com comportamento de classes abstratas (Figura 33). Novamente utilizamos o idioma *AspectJ Container Introduction* (Hanenberg et al., 2003).

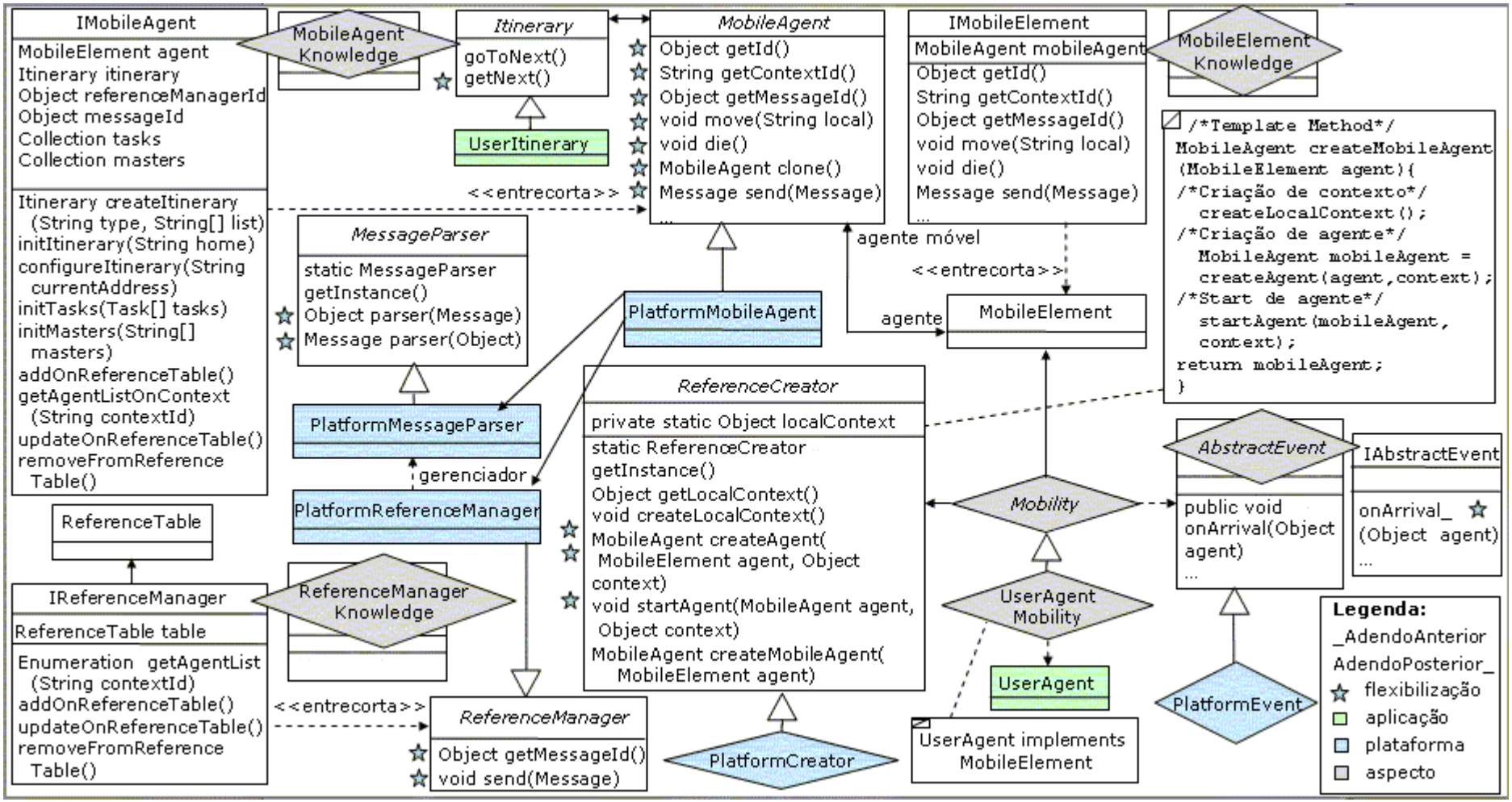

Figura 33. Pontos de Flexibilização de Plataformas no AspectM

6.4.3.
Dinâmica do Componente MobilityManagement

Na Figura 34, o protocolo de instanciação de agentes (Figura 21) é novamente ilustrado a fim de demonstrar a utilização das classes `ReferenceCreator` e `ReferenceManager` da Figura 33.

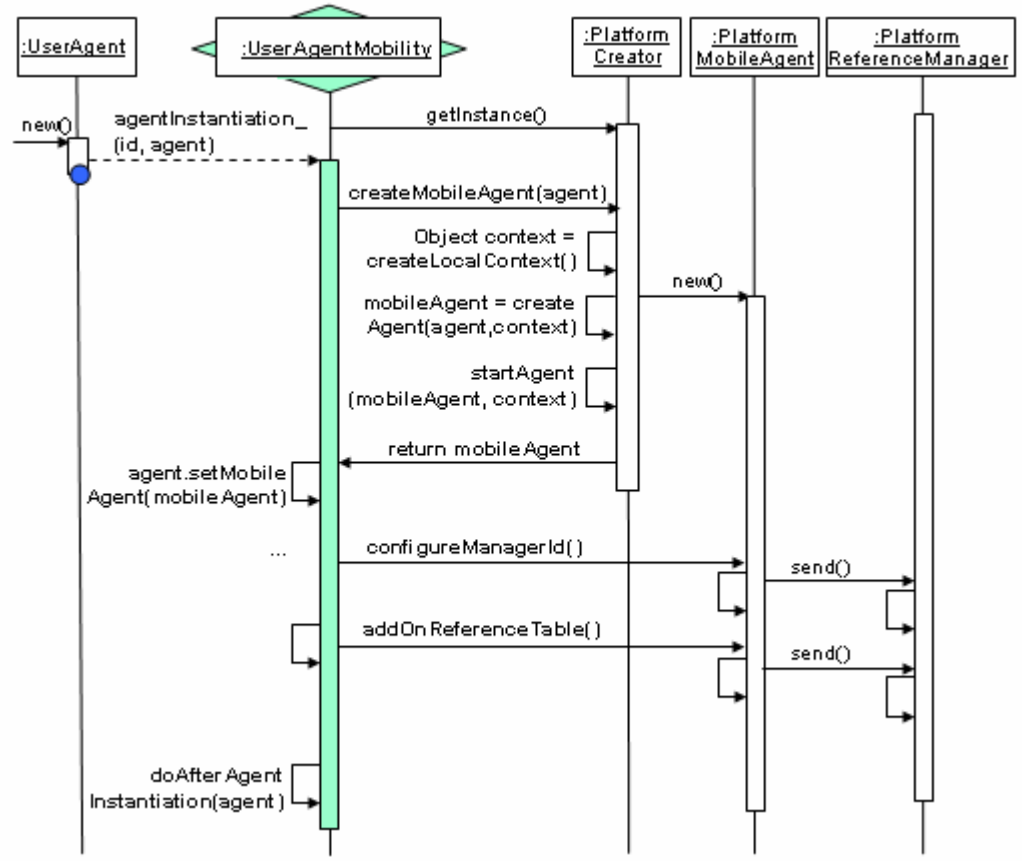

Figura 34. ReferenceCreator e ReferenceManager na Instanciação

Na Figura 34, não são mostrados os passos já definidos na Seção 6.2. Mais especificamente, antes da execução do método `createMobileAgent()`, o protocolo de instanciação realiza a mesma seqüência de passos já descrita na Figura 27: (1) obtém do usuário o identificador do agente móvel, e (2) obtém do usuário uma referência para o agente que requer serviços de mobilidade na aplicação. A diferença é que, na Figura 34, o método `createMobileAgent()` não é mais ilustrado como fazendo parte da interface do aspecto `Mobility`. A responsabilidade de criação de agentes móveis em plataformas é delegada à classe `ReferenceCreator`. As responsabilidades de `ReferenceCreator` são: (i) criar um contexto local para a execução de agentes móveis (`createLocalContext()`,

na Figura 34), (ii) instanciar os agentes móveis no contexto de execução anterior (`createAgent()`, na Figura 34), e (iii) tornar acessíveis os serviços de mobilidade para agentes móveis instanciados (`startAgent()`, na Figura 34).

Após a execução destes passos, o fluxo de execução é retornado para o adendo de instanciação do aspecto `Mobility`. A responsabilidade de manter uma associação unívoca entre o agente da aplicação e seu respectivo representante na plataforma é efetuada através dos métodos `createMobileAgent(agent)` e `agent.setMobileAgent()`. Após esta associação, a mesma seqüência de invocação de métodos da Figura 27 é mantida, a saber: `setId()`, `initItinerary()`, `initTasks()` e `initMasters()`.

Outra diferença entre as Figuras 27 e 34 surge no passo de manutenção da tabela de referências de agentes em uma aplicação. Na Figura 27, esta operação foi ilustrada através da invocação do método `addOnReferenceTable()`, como se a responsabilidade de atualização da tabela pertencesse ao agente móvel instanciado na plataforma. Entretanto, a invocação do método `addOnReferenceTable()` encapsula a seguinte seqüência de passos executada pelo agente `UserAgent` na Figura 34:

- obter uma referência para o `ReferenceManager` da aplicação (método `configureManagerId()`). O `ReferenceManager` mantém atualizados atributos importantes para a consistência dos SMAs móveis, como o identificador de contexto de execução e o identificador para troca de mensagens. No *framework* AspectM, o `ReferenceManager` é implementado como um agente móvel que responde a solicitações dos agentes da aplicação através de troca de mensagens. Todo agente instanciado na plataforma deve manter uma referência para o `ReferenceManager` da aplicação;
- na execução do método `addOnReferenceTable()`, enviar uma mensagem para o `ReferenceManager` notificando a instanciação de novo agente móvel na plataforma.

Após estes passos, a Figura 34 volta a executar os mesmos procedimentos especificados na Figura 27.

A Figura 35 ilustra novamente o protocolo de inicialização de agentes, agora para demonstrar interações com a classe `ReferenceManager`.

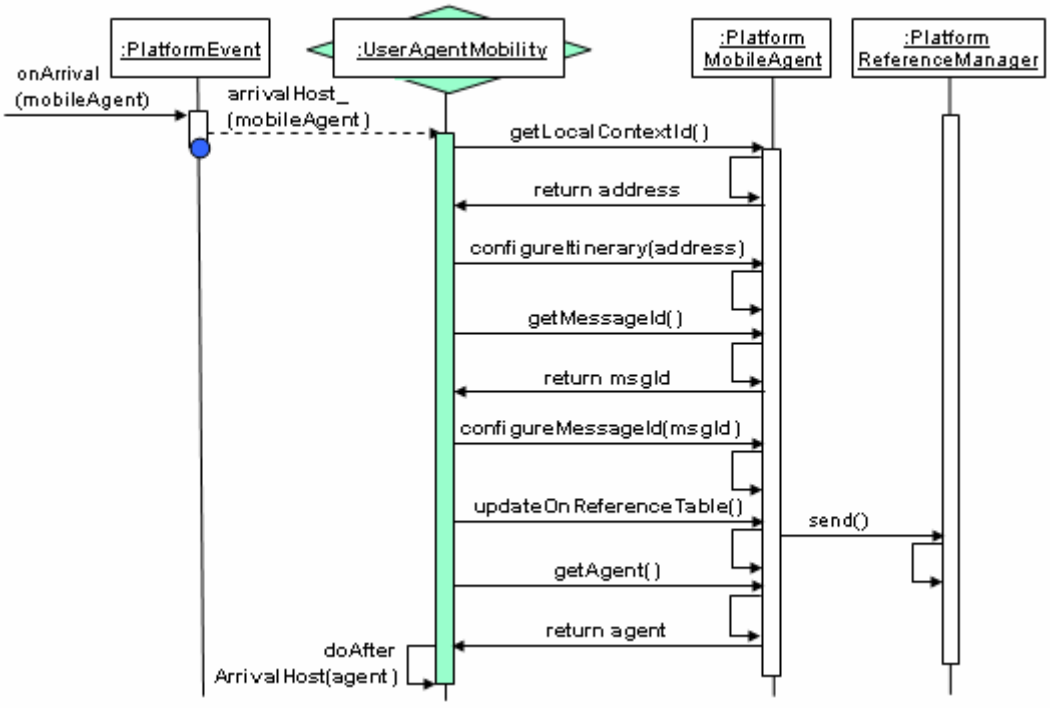

Figura 35. ReferenceManager no Protocolo de Inicialização do Agente

Na Figura 35, a mesma seqüência de procedimentos do protocolo de inicialização da Figura 29 é executada, até a execução do método `updateOnReferenceTable()`. Novamente, na Figura 29 a invocação de `updateOnReferenceTable()` foi ilustrada como se a responsabilidade de atualização da tabela pertencesse ao agente móvel instanciado na plataforma. Entretanto, a chamada ao método `updateOnReferenceTable()`, como também ocorre com `addOnReferenceTable()`, encapsula uma troca de mensagens entre o agente `UserAgent` e o `ReferenceManager`. Após a execução do método `updateOnReferenceTable()`, o `ReferenceManager` possui atualizados o identificador de contexto e para troca de mensagens dos agentes na aplicação.

Finalmente, a Figura 36 ilustra o protocolo de destruição de agentes a fim de demonstrar a interação com a classe `ReferenceManager`. A execução do método `removeFromReferenceTable()` exclui a referência ao agente não mais acessível na plataforma, após a invocação do método `die()`.

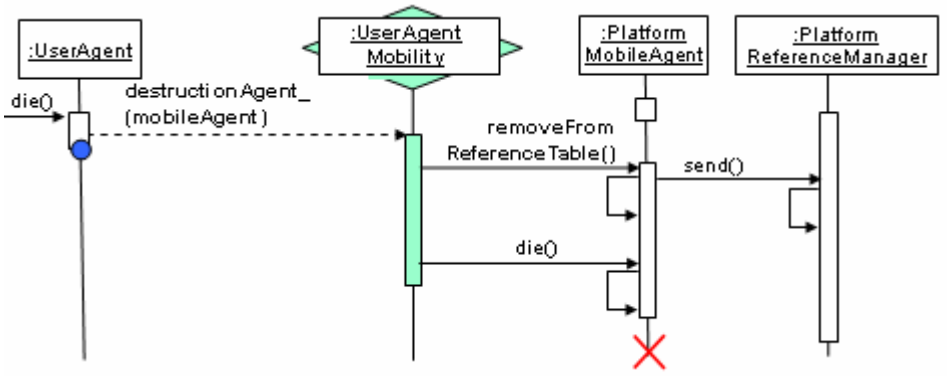

Figura 36. ReferenceManager no Protocolo de Destruição do Agente

6.4.4. Processo de Instanciação de Pacote de Plataforma

Supondo que a implementação do pacote referente à plataforma escolhida ainda não está disponível, os usuários podem concretizar os pontos de flexibilização do *framework* AspectM, seguindo uma ordem crescente de complexidade: (1) `MessageParser`; (2) `MobileAgent`; (3) `AbstractEvent`; (4) `ReferenceCreator`; (5) `ReferenceManager`. Todos os pontos de flexibilização nas classes enumeradas devem ser concretizados pelo usuário de AspectM.

Após a especificação destes pontos de flexibilização, que tornam possível a comunicação do AspectM com uma plataforma de mobilidade específica, o processo de instanciação de uma aplicação segue a mesma seqüência de passos descrita na Seção 6.3.3.