5
A linguagem de padrões

"We must start decomposing a system by objects, and then use the resulting structure as the framework for expressing the other perspective". (Booch)

Com a evolução do desenvolvimento de SMAs, os engenheiros de software vão enfrentando novos problemas para resolver a separação de concerns do agente. De fato, apesar de essa separação ser crítica para os desenvolvedores de SMAs, alcançá-la em SMAs reais não é algo trivial. Dependendo da complexidade do agente, um aspecto do agente afeta não apenas as classes kernel, como também outros aspectos do agente (Seção 3.6.4). O capítulo anterior apresentou aos projetistas de SMAs as diretrizes básicas para a definição de concerns do agente que devem ser separados e a ordem em que devem ser tratados. Entretanto, o método arquitetural não oferece diretrizes detalhadas sobre como estruturar cada aspecto do agente. O refinamento do método proposto no capítulo anterior levanta algumas questões importantes, como:

(i) Como definir a estrutura interna de cada aspecto do agente?

(ii) como definir os diferentes relacionamentos entre classes kernel e aspectos do agente?

(iii) com refinar os aspectos do agente no contexto de diferentes papéis e tipos de agente?

(iv) como compor vários aspectos e resolver conflitos de composição entre eles?

(v) como implementar os aspectos do agente usando uma linguagem orientada a aspectos?

Neste capítulo, é apresentada uma linguagem de padrões a fim de responder a essas questões, fornecendo diretrizes mais específicas para cada concern do agente. Os padrões podem ser vistos como minimétodos para lidar com as especificidades dos aspectos do agente. A linguagem de padrões orienta em detalhes *como ir de*

objetos a agentes usando aspectos. Os padrões orientados a aspectos desacoplam as propriedades de agente da funcionalidade básica dos “objetos de agente”.

De fato, os padrões de projeto são um veículo importante para o suporte à construção de sistemas orientados a aspectos com reusabilidade e manutenibilidade (Seção 2.1). A necessidade de padrões de projeto é ainda maior para o desenvolvimento de software orientado a aspectos uma vez que esse paradigma oferece muitas formas de decompor e compor um sistema de software (Seção 2.2.3). Apesar de o paradigma orientado a aspectos defender o suporte a uma melhor separação de *concerns*, ainda não está claro como alcançar isso em termos de abstrações orientadas a aspectos. O desafio fica ainda maior ao usar aspectos para crosscutting concerns específicos a domínios, como concerns do agente. No entanto, conforme mencionado anteriormente (Seção 2.2.3), experiências com boa prática em desenvolvimento de software orientado a aspectos são bastante limitadas e ficaram restritas a crosscutting concerns comuns ou triviais, como auditoria [12, 88], rastreamento [12, 88], distribuição [226, 88], persistência [226, 88], concorrência [141] e tratamento de exceções [87, 97, 161]. Além disso, eles foram quase sempre estudados de forma isolada. Há pouca compreensão sobre o uso de aspectos com vários concerns em um sistema de software.

A linguagem de padrões proposta guia os projetistas em relação a como estruturar os vários concerns do agente nos níveis do projeto detalhado e da implementação. A linguagem é composta por sete padrões: (1) o padrão Kernel do Agente, (2) o padrão Interação, (3) o padrão Adaptação, (4) o padrão Autonomia, (5) o padrão Papel, (6) o padrão Mobilidade e (7) o padrão Aprendizagem. O padrão Kernel do Agente é o padrão central da linguagem uma vez que todos os outros baseiam-se em sua estrutura e comportamento. Esse padrão define o objeto alvo que desejamos transformar em um agente, introduzindo as propriedades de agente. Ele também descreve as classes adicionais para representar o conhecimento do agente. Cada padrão define uma solução detalhada para introduzir uma propriedade de agente ao objeto alvo. A linguagem também mostra como juntar todos os padrões orientados a aspectos.

O primeiro alvo da linguagem de padrões é todos os engenheiros de software de aplicações multiagentes que precisam definir e implementar as diferentes propriedades de agente que regem seus sistemas. Também pode ser interessante para os desenvolvedores de diferentes tipos de frameworks de SMAs (Seção 3.7) usar os padrões propostos, uma vez que podem decidir incorporar as soluções orientadas a aspectos diretamente em suas arquiteturas.

As contribuições deste capítulo foram parcialmente descritas nos trabalhos [90]. No entanto, esses trabalhos não apresentam em detalhes as soluções orientadas a aspectos propostas, porque seu foco é a apresentação do método arquitetural (Capítulo 4). Além disso, não são apresentados no formato de padrões. Apenas mostram uma versão preliminar das soluções de padrão apresentadas aqui. Os padrões surgiram a partir da aplicação extensiva do método em diferentes sistemas (Capítulo 7) e sua implementação em duas linguagens orientadas a aspectos diferentes, AspectJ e Hyper/J. O estudo comparativo da implementação das duas linguagens pode ser encontrado nos trabalhos [42, 260]. Esses trabalhos mostram como a implementação dos padrões orientados a aspectos no sistema Portalware (Capítulo 4) pode ser facilmente mapeada em uma implementação de Hyper/J (Capítulo 7). A linguagem de padrões também foi publicada como um relatório técnico e enviada a *ACM Transactions on Software Engineering and Methodology (TOSEM)*. Alguns relatórios técnicos e trabalhos apresentados em workshops documentam as diferentes fases do desenvolvimento da linguagem de padrões [81, 84, 94].

O restante deste capítulo está organizado da seguinte forma. A Seção 5.1 apresenta o exemplo usado ao longo do capítulo para ilustrar os padrões de projeto. A Seção 5.2 apresenta uma visão geral da linguagem de padrões orientada a aspectos. As Seções de 5.3 a 5.9 apresentam os padrões de projeto. A Seção 5.10 discute as questões de implementação e implantação. Já a Seção 5.11 introduz os trabalhos relacionados. A Seção 5.12 apresenta o resumo do capítulo.

5.1
Expert Committee: um estudo de caso

Esta seção introduz um SMA a fim de ilustrar os padrões apresentados neste capítulo. Esse protótipo é um sistema de gerenciamento de conferência, um exemplo clássico de uma aplicação baseada em agentes de software [57, 253]. Esse sistema inclui vários concerns do agente, típicos de muitas aplicações orientadas por agente existentes. Ele é derivado de um estudo de caso realizado no Laboratório de Engenharia de Software na PUC-Rio no Brasil, doravante Expert Committee (EC). EC é um sistema multiagentes que oferece suporte ao gerenciamento de envios de trabalho e ao processo de revisão de uma conferência. Os agentes de software foram introduzidos ao sistema de EC a fim de auxiliar os pesquisadores nas atividades que consomem tempo nos envios de trabalho e processos de revisão.

Os agentes de EC são assistentes de software que representam os autores dos trabalhos, a chair, os membros do comitê do programa e os revisores e que coordenam suas atividades. São atribuídos diferentes papéis a cada usuário do EC, mas as principais são: (i) autor do trabalho, (ii) revisor, (iii) membro do comitê do programa e (iv) chair. O papel de chair possui planos para a distribuição de propostas de revisão; o papel de revisor e membro do comitê do programa têm planos para a avaliação das propostas da chair. A chair, os membros do comitê do programa e os revisores negociam entre si a realização das revisões. Há outros planos para tratar da carga de trabalho do usuário e de convites a novos revisores. O conhecimento de agente modela o mundo exterior a fim de refletir as preferências do usuário e o status do processo de revisão. O sistema EC também incorpora os agentes de informação. Cada tipo de agente oferece serviços diferentes, mas todos são interativos, adaptativos e autônomos. Cada tipo de agente possui também diferentes propriedades de agência específicas a aplicações.

Para simplificar, esta seção enfatiza a descrição dos agentes de usuário. Esses agentes possuem outras propriedades, incluindo papéis, aprendizagem e mobilidade. Os agentes de usuário interagem com o ambiente usando duas infra-estruturas de comunicação: JADE [20] e uma arquitetura blackboard [155]. O blackboard é usado para a comunicação interna entre os agentes, e a infra-estrutura JADE é usada para a

interação com agentes externos ao sistema. São usadas duas linguagens de comunicação: ACL [75] e uma linguagem de comunicação interna, também em conformidade com a especificação FIPA [75]. Os serviços de agente também podem ser acessados diretamente chamando as classes Agent. Os agentes de usuários monitoram os objetos do ambiente a fim de adaptar seu conhecimento e comportamento e aprender sobre as preferências dos usuários. Eles observam os eventos associados aos componentes de banco de dados, os objetos GUI e os componentes lógicos do negócio.

Esse comportamento sensorial e as instruções explícitas do usuário disparam a adaptação do comportamento e do conhecimento do agente. Os agentes aprendem as preferências do usuário ao observá-lo, observar os componentes da aplicação e a colaboração com outros agentes. Essa experiência de aprendizagem é indireta porque o agente criará seu conhecimento por meio dos resultados das negociações. O aprendizado automático é usado para tratar da aquisição de conhecimento. São usadas diferentes técnicas de aprendizagem no sistema EC: Temporal Difference Learning (TD-Learning) [172] e Least Mean Squares (LMS) [172]. TD-Learning é usada pelo papel de revisor a fim de aprender as preferências do usuário em relação aos assuntos que gosta de revisar. LMS é usado pela chair para aprender sobre as preferências do revisor. Este capítulo mostra os diferentes cenários que envolvem cada concern do agente e por que o projeto orientado a objetos para propriedades de agente resulta em crosscutting concerns. Ele também apresenta como usar as soluções dos padrões para modularizar os crosscutting concerns do agente. O sistema EC foi escolhido para ilustrar os padrões de projeto propostos neste capítulo porque é mais complexo do que o sistema Portalware (Seção 4.1). Além disso, esse exemplo envolve cenários interessantes para explorar todas as soluções de padrões. A Figura 37 descreve a arquitetura orientada a aspectos dos agentes de usuário do EC. Ela mostra a natureza crosscutting dos concerns do agente. Essa arquitetura é mais complexa do que as arquiteturas de agente do sistema Portalware em vários sentidos: (i) inclui todas as propriedades adicionais e de agência, (ii) possui uma interface sensorial para observar eventos externos nos objetos do ambiente, (iii) também incorpora a interface crosscutting LearningKnowledge, (iv) incorpora diferentes técnicas de aprendizagem

e (v) os papéis chair e revisor são associados às propriedades adicionais e de agência – assim, precisam ser definidos aspectos adicionais e de agência para esses papéis.

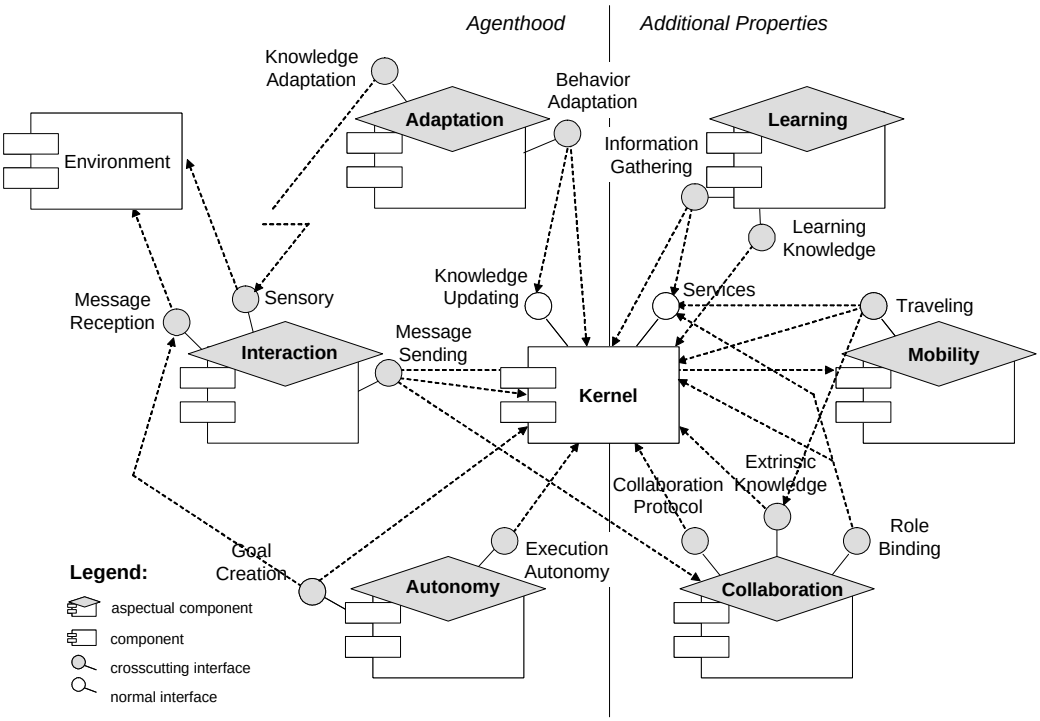


Figura 37. A arquitetura orientada a aspectos dos agentes do EC.

5.2
A linguagem de padrões: uma visão geral

O problema tratado pela linguagem de padrões envolve o projeto detalhado de agentes de software, enfatizando a separação de concerns. Elas se baseiam em aspectos para separar as propriedades de agente e permitem a composição flexível na construção de agentes heterogêneos (Seção 5.2.1). Cada padrão possui um propósito especial e trata de uma propriedade de agente específica. A linguagem de padrões contém informações conectivas úteis que ajudam a validar os padrões e a aplicá-los (Seção 5.2.3).

5.2.1
O propósito

O propósito básico da linguagem de padrões é oferecer suporte à modularização de crosscutting concerns do agente. Ela forma a base para a separação e a composição de concerns do agente, oferecendo o desenvolvimento de agentes com manutenibilidade e reusabilidade. O propósito não é fornecer soluções para o mecanismo usado para implementar cada concern do agente. Cada concern do agente requer mecanismos de implementação específicos e uma linguagem de padrões adicional. Por exemplo, o padrão de Lange [10] trata de alguns problemas específicos relacionados à mobilidade do agente. Nesse sentido, os padrões propostos neste capítulo podem estar conectados a outros padrões que tratam de problemas mais específicos para um dado concern. Eles apresentam soluções orientadas a aspectos para a segregação de concerns do agente.

5.2.2
Por que são padrões de projeto?

As soluções propostas aqui são padrões por algumas razões. Primeiro, os padrões foram identificados e abstraídos com base em sua aplicação em três SMAs em diferentes domínios de aplicação por vários grupos de desenvolvedores [51, 57, 89] (Capítulo 7), e no estudo de algumas arquiteturas de implementação (Seção 3.7). Como conseqüência, podem ser aplicados a uma grande variedade de aplicações de agente. Segundo, apesar de os padrões serem originais, uma vez que usam soluções orientadas a aspectos, o uso de aspectos dedica-se a resolver os problemas associados a padrões orientados a objetos conhecidos para o desenvolvimento de concerns do agente (Seção 3.6). As forças desses padrões enfatizam a reusabilidade e manutenibilidade. Por exemplo, o padrão Papel melhora a solução do padrão Objeto do Papel [19], que é a melhor solução conhecida para a estruturação de papéis do agente [136, 146]. O padrão Interação melhora a solução do padrão Adaptador [79], que é uma solução orientada a objetos elegante para a modularização do concern de interação [137].

Além do mais, os padrões foram implementados em duas linguagens orientadas a aspectos diferentes (Seção 5.11). A implementação do padrão refina alguns idiomas e padrões de aspectos gerais [114, 115]. Como a linguagem de padrões é independente do framework de implementação de SMAs, muitos desenvolvedores desse tipo de sistema poderão empregá-lo.

5.2.3
A estrutura da linguagem de padrões

A linguagem é composta por sete padrões de projeto classificados em dois tipos: padrões de agência e padrões adicionais. A Figura 38 ilustra a estrutura da linguagem de padrões. Um retângulo denota um padrão, com as setas representando os relacionamentos entre os padrões. Um círculo representa o padrão Kernel do Agente que oferece o contexto no qual todos os demais padrões agem. A figura também descreve os relacionamentos crosscutting entre os padrões de projeto. Apesar de um relacionamento crosscutting representar dois tipos de crosscutting - crosscutting estático e crosscutting dinâmico (Seção 2.2), a Figura 38 não apresenta essa distinção. A descrição de cada padrão esclarece quando cada tipo de crosscutting é usado. Cada padrão depende dos padrões que afeta; a rede dessas conexões entre os padrões é que cria a linguagem.

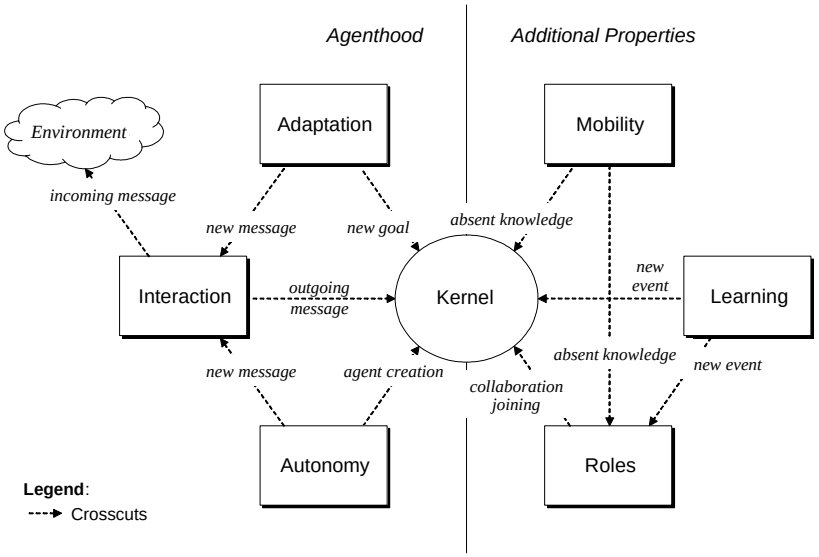


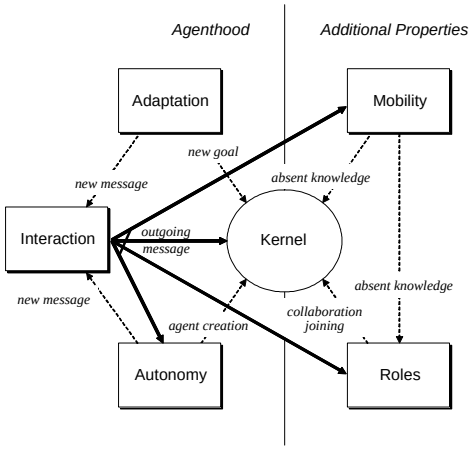
Figura 38. A estrutura da linguagem de padrões.

Cada padrão de projeto está relacionado a mais de um padrão. Por exemplo, o padrão Interação afeta o padrão Kernel e os objetos do ambiente. Quando há uma mensagem saindo de planos de agente ou ações de agente (padrão Kernel do Agente), o padrão Interação é responsável pela detecção de eventos externos e pelo envio e recebimento de mensagens. Quando há uma mensagem chegando das classes do ambiente, o padrão Interação é responsável pelo recebimento de mensagens de forma que o agente possa tratá-las e interpretá-las. Por outro lado, o padrão Adaptação e Autonomia afetam o padrão Interação sempre que chega uma nova mensagem. A nova mensagem pode resultar na adaptação de conhecimento (padrão Adaptação) e na criação de objetivo (padrão Autonomia).

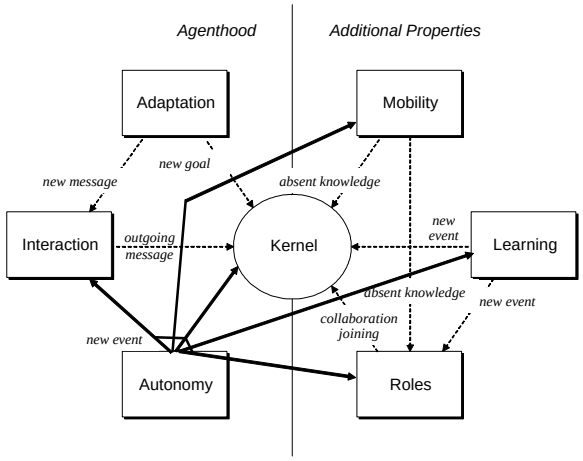
A Figura 38 apresenta apenas os inter-relacionamentos de padrões básicos, ou seja, os relacionamentos dependentes da aplicação. Esses relacionamentos sempre ocorrem e são independentes da complexidade do agente. No entanto, muitos relacionamentos crosscutting aumentam com o aumento da complexidade do agente. Isso é uma conseqüência da natureza crosscutting dos concerns do agente (Seção 3.6.4). A Figura 39 mostra relacionamentos crosscutting adicionais para o padrão Interação (39a) e o padrão Autonomia (39b), os dois aplicados ao exemplo do EC (Seção 5.1). Nesse sistema, a Figura 39a ilustra que as mensagens são enviadas a partir de diferentes componentes do agente: componentes de papel (padrão Papel), componentes de mobilidade (padrão Mobilidade) e planos de decisão (padrão Autonomia). A Figura 39b mostra que o comportamento autônomo de agentes cognitivos do EC afeta elementos de cinco padrões.

Os padrões de projeto são estruturados seguindo os modelos usados na literatura [35, 79] (Seção 2.1.2). Apesar de a inclusão de variantes e usos conhecidos no modelo do padrão parecer muito extensa, suas descrições foram incluídas para colocar este trabalho de pesquisa em perspectiva, levando em consideração outras contribuições encontradas na literatura. Os padrões são expressos usando a notação descrita na Seção 2.2. Omitimos a seção de implementação de cada padrão com o objetivo de simplificar sua apresentação. Contudo, introduzimos a seção de implementação dos padrões no Apêndice I, porque sua implementação em AspectJ também é uma

contribuição deste trabalho. Só estão disponíveis na literatura experiências usando crosscutting concerns bem conhecidos, como a distribuição, a persistência e a transação. O Apêndice I descreve trechos do código relevantes para a compreensão dos benefícios do padrão no nível da implementação. Além disso, discute questões importantes sobre a implementação de padrões usando AspectJ, uma linguagem orientada a aspectos atualmente em voga.



(a) O padrão Interação



(a) O padrão Autonomia

Figura 39. Vários padrões crosscutting.

5.3
O padrão Kernel do Agente

Intenção. O padrão Kernel do Agente descreve a estrutura básica dos tipos de agente; define as classes para representar o conhecimento intrínseco dos tipos de agente.

Também conhecido como. *Padrão Fundação, Padrão Esqueleto*.

Contexto. Os projetistas de SMAs precisam especificar as funcionalidades básicas dos tipos de agente. Cada tipo de agente possui seu próprio conhecimento intrínseco (Seção 3.2.3). Eles precisam definir o conhecimento intrínseco dos tipos de agente de forma que fique separado do conhecimento extrínseco. Os elementos fundamentais que expressam o conhecimento do agente são suas crenças, objetivos, ações e planos (Seção 3.3.1).

Exemplo de motivação. Os agentes do usuário no sistema EC possuem o conhecimento extrínseco e intrínseco. Em relação ao conhecimento intrínseco, cada agente possui crenças, objetivos, ações e planos básicos. Todos os agentes possuem crenças genéricas, incluindo seu identificador, seu nome e a lista de agentes disponíveis no ambiente. Os agentes do usuário têm crenças intrínsecas, incluindo o currículo, uma lista de trabalhos e a lista de interesses de pesquisa dos usuários. Eles têm objetivos intrínsecos, como o objetivo de atualizar o currículo do usuário em diferentes instituições de pesquisa à medida que ele vai mudando.

Em relação ao conhecimento extrínseco, o papel de chair introduz algum conhecimento adicional aos agentes do usuário. Ele inclui conhecimento específico, como os prazos da conferência, a lista de trabalhos enviados, a lista de revisores e seus agentes associados etc.

Problema. Como definir a estrutura básica dos tipos de agente do sistema? Muitas forças são associadas a esse problema de projeto:

- O projeto do agente deve oferecer suporte à definição de separação de conhecimento intrínseco e extrínseco. Além disso, esses tipos de conhecimento devem ser facilmente compostos.
- É necessária uma solução flexível e reutilizável para facilitar a reutilização de diferentes partes do conhecimento. Os desenvolvedores de SMAs devem ser capazes de entender e refinar com facilidade o conhecimento de agente básico.
- O conhecimento de agente deve ser separado da lógica de controle associada a diferentes propriedades de agente, como autonomia, adaptação e aprendizagem.
- A própria estrutura deve identificar um único componente que representa o agente no sistema.

Solução. Usar abstrações orientadas a objetos para representar a estrutura básica dos tipos de agente. Usar classes para representar um agente e seus elementos de conhecimento (Seção 4.4.1). A classe Agent especifica a estrutura principal de um agente e pode ser estendida para criar os tipos de agente da aplicação. Cada instância Agent representa um agente e o identifica de forma única no SMA. Os métodos

representam as ações de agente, enquanto os atributos representam as crenças do agente. Os atributos da classe Agent são usados para representar as crenças gerais, comuns a todos os tipos de agente na aplicação. Os métodos da classe Agent são usados para atualizar as crenças gerais e implementam as ações gerais do agente.

As classes também são usadas para representar os elementos do conhecimento de um agente (Figura 40). As subclasses Agent representam os tipos de agente (Seção 4.4.3). As subclasses Agent contêm referências aos elementos do conhecimento intrínseco, enquanto os aspectos contêm referências a elementos do conhecimento extrínseco. A Figura 40 apresenta as principais classes que representam o conhecimento do agente. A Figura 41 ilustra como criar tipos de agente; os aspectos são usados para introduzir o conhecimento extrínseco de forma transparente para o conhecimento intrínseco do agente.

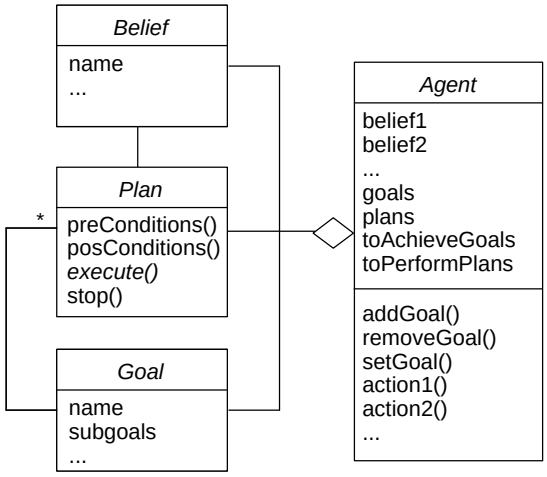


Figura 40. A visão estática do padrão Kernel do Agente.

Estrutura. Os atributos de um objeto Agent fazem referências a objetos que representam elementos do conhecimento intrínseco, a saber os objetos Belief, Goal e Plans. A classe Agent oferece operações padrão para a busca, inserção, atualização e exclusão dos elementos de conhecimento. A classe Agent também possui atributos para representar crenças genéricas do agente. Por exemplo, cada agente está acoplado a um nome, que é uma crença genérica do agente.

Os engenheiros de SMAs devem refinar a classe abstrata Agent para definir os tipos de agente do sistema (Figura 41). Para definir o conhecimento intrínseco de

tipos de agente, os projetistas de aplicação devem dividir as classes Belief, Goal e Plan em subclasses. Essas classes são estruturadas como hierarquias. A classe Belief pode ter os atributos a seguir: (i) um nome para identificar as instâncias da crença e (ii) atributos dependentes de aplicação. A classe Belief possui métodos de busca e atualização de crenças.

A classe Goal possui um nome e métodos set e get. Também possui uma lista de subobjetivos associados, uma vez que um objeto Goal pode ser decomposto em subobjetivos. Além disso, tem uma lista de planos porque um objetivo pode ter diferentes planos e, assim, um objeto Goal pode ter mais de um objeto Plan associado.

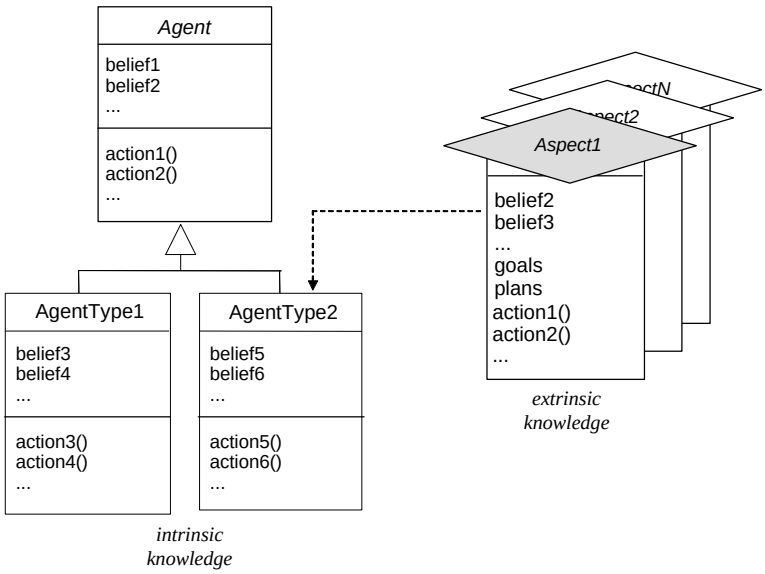


Figura 41. Definição dos tipos de agente.

A classe Plan tem um nome de identificação e um método abstrato execute() que deve ser chamado para dar início a um plano. Essa classe genérica pode ser implementada como uma thread. Uma subclasse de Plan deve implementar o método execute() de acordo com o plano concreto. Em geral, os planos são implementados como uma seqüência de ações. A classe Plan possui um método stop() que pode ser chamado para conclui o plano. Ela também define os métodos para verificar as precondições e definir as pós-condições. As *precondições* listam as crenças que

devem ser mantidas a fim de executar o plano, enquanto as *pós-condições* descrevem os efeitos da execução de um plano bem-sucedido usando crenças de um agente (Seção **3.3.1**).

Variantes. *Kernel de Herança*. A solução do padrão define como estruturar o conhecimento do agente desde o início. Nessa solução, as referências a elementos do conhecimento são definidas nas classes Agent que representam os tipos de agente. Diferente dessa solução, a variante *kernel de herança* incorpora o uso de aspectos para introduzir de forma transparente os elementos de conhecimento a um objeto existente, como listas de objetivos e planos. A introdução desses elementos é especificada nas inter-type declarations.

Conseqüências. O uso do padrão Kernel do Agente oferece as vantagens a seguir:

- *Adaptação dinâmica*. A representação explícita e a manipulação de objetivos e planos facilitam, por exemplo, um "ajuste" em tempo de execução do comportamento do sistema necessário para lidar com as diferentes circunstâncias.
- *Reutilização do conhecimento.* A modularização de planos em classes oferece suporte a sua reutilização em diferentes contextos como parte de outros planos.
- *Separação de concerns.* O conhecimento do agente deve ser separado da lógica de controle associada a diferentes propriedades de agente, como autonomia (Seção 5.5), adaptação (Seção 5.6) e aprendizagem (Seção 5.9). Não há referência a essas propriedades nas classes kernel agent. Além disso, o conhecimento intrínseco é definido separado do conhecimento extrínseco.
- *Reutilização da Lógica de Controle*. A separação da lógica de controle e os algoritmos de princípios do conhecimento permitem escrever procedimentos de inferência reutilizáveis e otimizados [22, 57].
- *Reutilização de Ações*. As ações básicas são desacopladas dos planos de forma que as ações possam ser reutilizadas no contexto de diferentes planos.

- *Identificação Única*. Um objeto único, uma instância de Agente, representa cada agente. Os aspectos do agente são trançados dentro das classes Agent com base no mecanismo dos processos de combinação (Seção 2.2.2).
- *Transparência*. A variante *Kernel de Herança* define como introduzir de forma transparente os elementos de conhecimento em um objeto existente, que desejamos transformar em agente.

Usos conhecidos. Essa abordagem tem o suporte de alguns frameworks de implementação OO. Por exemplo, a arquitetura SkeletonAgent [37] usa classes para estruturar planos, objetivos e crenças do agente.

Padrões relacionados. O padrão Composição [79] pode ser usado para estruturar hierarquias de objetivo complexas (por exemplo, objetivos e subobjetivos) e hierarquias de crença complexas (por exemplo crenças compostas e primitivas). O padrão Estratégia [79] pode ser usado para estruturar diferentes implementações de plano. O padrão Papel (Seção 5.7) define como especificar o conhecimento extrínseco de tipos de agente. Todos os demais padrões orientados a aspectos desse catálogo estão relacionados ao padrão Kernel do Agente.

5.4
O padrão Interação

Intenção. O Padrão Interação modulariza o *comportamento interação*. Ele desacopla o comportamento interativo do agente de sua funcionalidade básica e dos objetos do ambiente.

Também conhecido como. *Padrão Comunicação, Padrão Sensorial*.

Contexto. O comportamento interação de um agente de software caracteriza-se pelo envio e recebimento de mensagens e pelo comportamento sensorial (Seção 3.3.2). Desejamos separar o comportamento de interação do kernel do agente.

Exemplo de motivação. No sistema EC, a interação é necessária em várias circunstâncias. As mensagens devem ser enviadas a partir de diferentes planos e

ações. Além disso, os agentes de usuário precisam enviar mensagens, por exemplo, quando estão exercendo o papel de revisor ou chair. A chair envia mensagens com as propostas de revisão aos revisores e eles enviam as respostas de volta. Cada revisor usa uma linguagem de comunicação diferente. Além do mais, um agente, como um revisor, precisa observar os diferentes componentes externos do software: (i) a agenda do usuário que é implementada por um sistema de software de programação de tarefas, (ii) a interface com o usuário e (iii) um objeto persistente que armazena o currículo do usuário. A Figura 42 ilustra um exemplo do comportamento interativo dos agentes do EC.

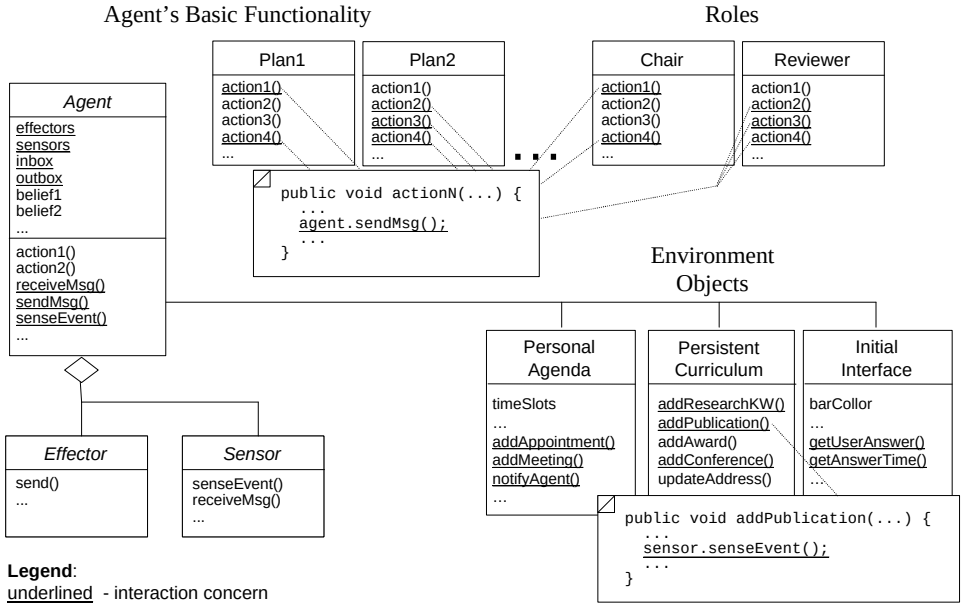


Figura 42. O projeto orientado a objetos do concern de interação.

As implementações e os projetos orientados a objetos [20, 137] da propriedade de interação são intrusivos. O comportamento interação afeta as classes kernel, os papéis e os objetos do ambiente. A Figura 42 ilustra como o comportamento interação está entrelaçado e espalhado pelas classes do agente e do ambiente. O método e o código afetados pela propriedade de interação estão sublinhados na figura. As classes kernel (Seção 5.3), que representam a funcionalidade básica do agente, são normalmente interligadas a métodos de interação, como sendMsg(), receiveMsg() e senseEvent() e atributos de interação,

como a caixa de entrada e saída. Além do mais, a implementação da interação afeta várias ações e planos de agente, que enviam mensagens para agentes colaborativos. O comportamento interação também afeta os métodos de papel, que são interligados às chamadas ao método sendMessage(). Ela também afeta muitos objetos do ambiente que são monitorados pelo agente (comportamento sensorial). Os objetos precisam notificar os agentes sobre os eventos relevantes chamando os métodos das classes Sensor.

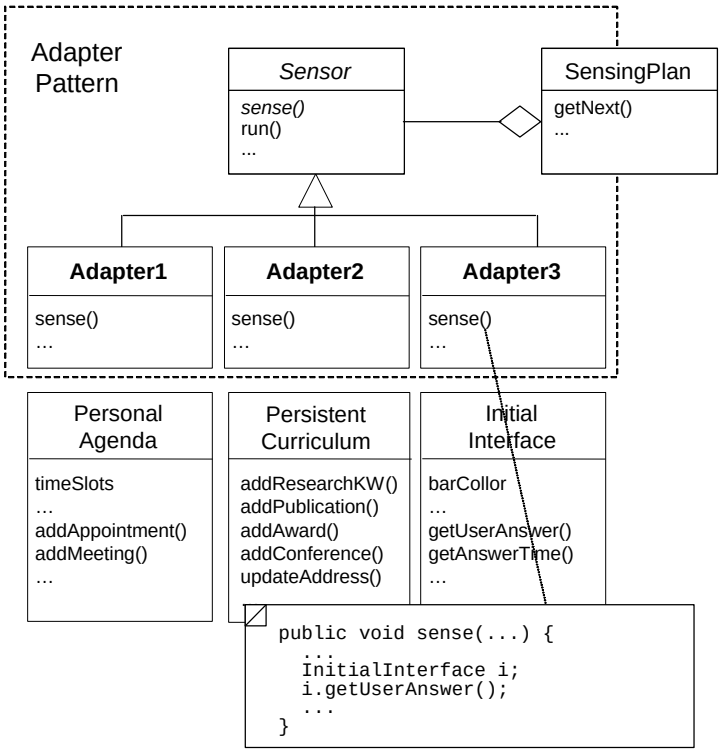


Figura 43. Comportamento detecção nos agentes do EC.

Kendall [137] propõe a realização do padrão Adaptador [79] para melhorar a separação do comportamento sensorial. Esse padrão é usado para criar uma classe reutilizável que coopera com as classes do ambiente. A Figura 43 ilustra a instanciação de padrões do sistema de EC. A principal abstração da solução de Kendall é um conjunto de Sensores específicos que precisam cooperar com as classes específicas a domínios. Essa solução fornece a interface do Sensor (o método

Sense()), enquanto o desenvolvedor da aplicação oferece SensorAdapters específico a domínios.

Há uma lista de objetos externos que precisam ser detectados; eles são armazenados na classe SensingPlan. A SensingPlan pode ser apenas um conjunto de sensores e suas mensagens relevantes. A Figura 44 mostra como um objeto Sensor que utiliza um Plano de Detecção e um SensorAdapter para detectar eventos externos levando em consideração as informações mutáveis em um objeto do ambiente. Ele obtém informações sobre novas preferências e compromissos do usuário. Para obter mais detalhes sobre essa solução de padrões, consulte [137].

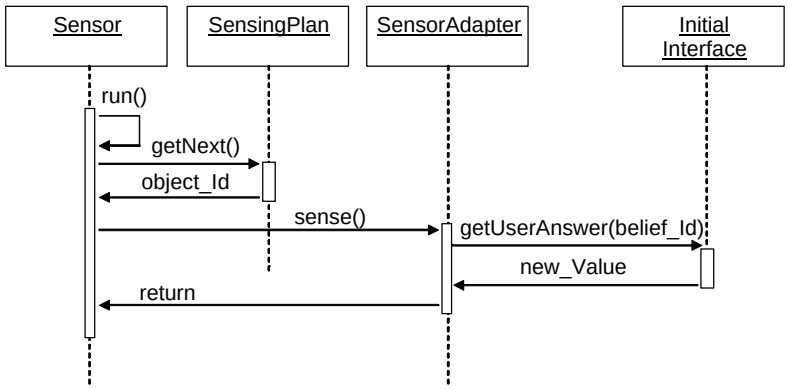


Figura 44. A visão dinâmica do padrão Adaptador.

A solução melhora a separação de concerns, uma vez que isola o comportamento sensorial nas subclasses Adapter e na classe SensingPlan. Essa estrutura leva a um código menos entrelaçado uma vez que o código de não-interação não se entrelaça com o código de interação, mas não o evita completamente. O código para o envio de mensagens dentro das classes Agent não pode ser desentrelaçado. Isso acontece porque a solução somente tenta resolver o problema relacionado ao comportamento sensorial. Ela não modulariza o comportamento interativo que está entrelaçado às classes kernel do agente. Ademais, nos casos em que pode ser desentrelaçado (comportamento sensorial), o projetista tem de pagar um preço alto por isso: adaptadores devem ser escritos apenas para cuidar das funções de detecção.

Além disso, a própria proposta baseada no adaptador introduz outros problemas. Cada sensor é uma thread que trabalha como um mecanismo de *pooling* que aguarda até que algumas informações sejam reunidas do ambiente externo. Esse monitoramento é caro e desnecessário, uma vez que as alterações no ambiente não podem ocorrer com freqüência. O comportamento detecção só deve ser ativado quando acontece alguma alteração no ambiente. Finalmente, a solução assume que há métodos públicos nas classes específicas a domínios que fornecem os serviços e as informações necessários. Essa premissa nem sempre é verdadeira em determinadas aplicações.

Problema. A implementação básica de planos do agente e ações associadas é interligada com chamadas a métodos específicos a interações que respondem ao envio de mensagens a outros agentes. Além disso, as funcionalidades básicas das classes do ambiente são alteradas a fim de incorporar as chamadas a métodos de detecção de forma intrusiva. Como conseqüência, o ambiente e os componentes kernel do agente são altamente acoplados ao comportamento interação. A estrutura básica do agente deve estar ciente do comportamento interação. As classes Environment não são projetadas para trabalhar como sensores e não devem conter referências explícitas a sensores. Como desacoplar o comportamento interação da estrutura básica dos agentes? As forças a seguir são associadas a esse problema:

- os engenheiros de SMAs devem definir a interação do agente de uma forma que tenha reusabilidade e manutenibilidade;
- a explosão no número de componentes do agente deve ser evitada;
- as classes externas não devem ser alteradas na implementação da capacidade sensorial do agente.

Solução. Usar aspectos para melhorar a separação do comportamento interativo do agente. Os aspectos Interaction são usados para capturar o comportamento de interação que afeta muitas partes de um agente de software. Eles separam o comportamento interação de elementos de conhecimento do agente, como ações e planos, e dos componentes de ambiente e papéis. Em outras palavras, os aspectos são

usados não só para modularizar o centro do comportamento interação, mas também para isolar todo o comportamento relacionado ao concern de interação, que inclui o envio e o recebimento de mensagens e o comportamento sensorial.

Ao usar os aspectos Interaction, definimos como os planos e as ações do agente interagem com o ambiente externo. Esses aspectos conseguem afetar alguns pontos de execução do agente – por exemplo chamadas a métodos nas classes Plan – e alteram sua execução normal a fim de enviar mensagens. Os aspectos Interaction monitoram esses pontos de execução a fim de identificar quando uma mensagem deve ser enviada usando classes auxiliares. As classes auxiliares são usadas para implementar sensores e efetores.

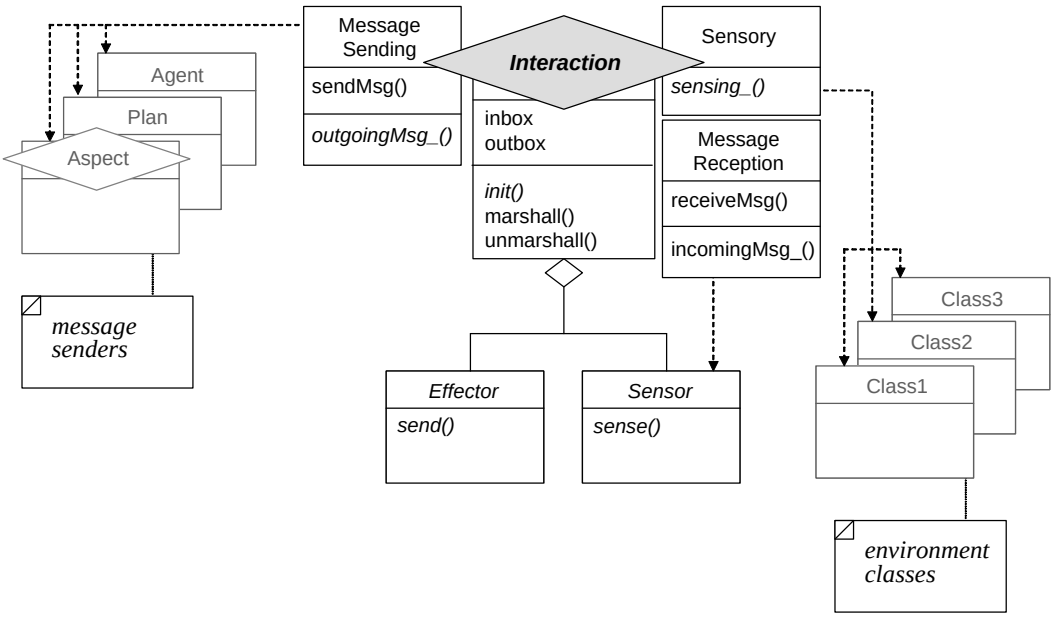


Figura 45. A visão estática do padrão Interação.

Estrutura. O padrão Interação possui três participantes principais e dois tipos de clientes:

Principais Participantes:

- **Aspecto Interaction**

- define a lógica básica do concern de interação.

- **Sensor**

- colabora com as classes Environment e as infra-estruturas de comunicação a fim de receber mensagens e eventos externos relevantes ao agente.

- **Efector**

- colabora com as classes Environment e as infra-estruturas de comunicação a fim de enviar mensagens e chamar métodos nas classes environment.

Participantes cliente:

- **Emissor da mensagem**

- fornece o contexto a partir do qual algumas mensagens precisam ser enviadas e não possui qualquer código de interação – pode ser um plano, um agente, ou um aspecto do agente.

- **Classe Environment**

- fornece o contexto a partir do qual os eventos precisam ser detectados – essa classe não possui código de interação.

A Figura 45 apresenta as partes comuns a todas as instanciações potenciais do padrão e outras partes que não são específicas a cada instanciação. As partes comuns são:

1. A existência de sensores e efetores (ou seja, o fato de algumas classes agirem como sensores e outras como efetores).
2. A lógica de interação geral:
    a. as mensagens são detectadas, desordenadas e armazenadas.
    b. as mensagens são armazenadas ordenadas e propagadas para o ambiente.
3. A interface para enviar e receber mensagens.

As partes específicas para cada instanciação do padrão são:

1. Quais classes são sensores e quais são efetores no contexto de um tipo de agente ou funções específicas.
2. Um conjunto de *join points* (Seção 2.2.2) em que as mensagens devem ser enviadas ao ambiente externo.

3. Um conjunto de join points nas classes do ambiente em que os eventos precisam ser detectados.

O propósito do aspecto Interaction é tornar interativas as instância de Agente. Em outras palavras, o aspecto Interaction estende o comportamento da classe Agent para enviar e receber as mensagens. Esse aspecto envia e recebe mensagens e detecta as alterações do ambiente por meio de sensores e efetores. As inter-type declarations (crosscutting estática) são usadas para adicionar os novos métodos específicos a interações: os métodos receiveMsg() e sendMsg(). As classes Sensor e Effector representam respectivamente os sensores e efetores e cooperam com as classes do ambiente. Os sensores e os efetores são classes e seu isolamento do aspecto pretende melhorar a reutilização.

O aspecto Interaction possui quatro partes: o próprio aspecto e três interfaces crosscutting. Ele mantém uma caixa de entrada, uma caixa de saída, métodos de inicialização, métodos para ordenar e desordenar as mensagens e define a lógica de interação. Como o aspecto Interaction implementa a lógica de interação, ele afeta a classe Agent, os sensores, efetores, as classes Agent ou Plan que precisam enviar uma mensagem para outros agentes, e as classes do ambiente que precisam ser observadas pelo agente.

As interfaces crosscutting definem como o aspecto Interaction afeta diferentes classes do sistema multiagentes. A interface MessageSending define um pointcut outgoingMsg que especifica os emissores de mensagens; ela especifica join points nas classes Agent a partir das quais as mensagens precisam ser enviadas para o mundo exterior. Alguns exemplos de join points são métodos de classes Plan e classes Agent. Observe que o pointcut outgoingMsg é abstrato porque os join points dependem das funções e dos tipos de agente específicos. O pointcut é concretizado nos subaspectos Interaction. A interface contém um advice que executa depois de execuções de ações em classes do agente, ações em classes do plano, outros aspectos associados a agentes (por exemplo, os aspectos de papel – Seção 5.7, aspectos de mobilidade – Seção 5.8). O propósito do advice é capturar as informações necessárias para enviar a mensagem aos agentes e atualizar a caixa de saída.

A interface MessageReception define um pointcut incomingMsg para interceptar execuções do método sense() nas classes Sensor; o objetivo é detectar a chegada de mensagens. Esse pointcut está associado a um after advice responsável pelo processamento de mensagens recebidas e pela atualização da caixa de entrada. A interface sensorial implementa o pointcut de detecção que declara quais métodos das classes do ambiente precisam ser monitoradas. O sensing_ advice processa os eventos externos e atualiza a caixa de entrada. O pointcut de detecção também é declarado como abstrato porque os join points dependem dos papéis e dos tipos de agente específicos.

A Figura 46 ilustra a instanciação de padrões do agente de usuário no sistema Expert Committee. O aspecto Interaction afeta diferentes aspectos do agente e classes nesse sistema, cerca de 15 componentes. Além disso, ele tem dois subaspectos: o aspecto ReviewerInteraction e o aspecto ChairInteraction. No entanto, para fins de simplificação, ela só apresenta algumas dessas classes e aspectos. Ela mostra uma classe Plan, uma classe Agent e um agente de Mobilidade. Os outros seguem essencialmente o mesmo padrão. Também foram omitidas as classes auxiliares, que oferecem suporte à tradução de mensagens.

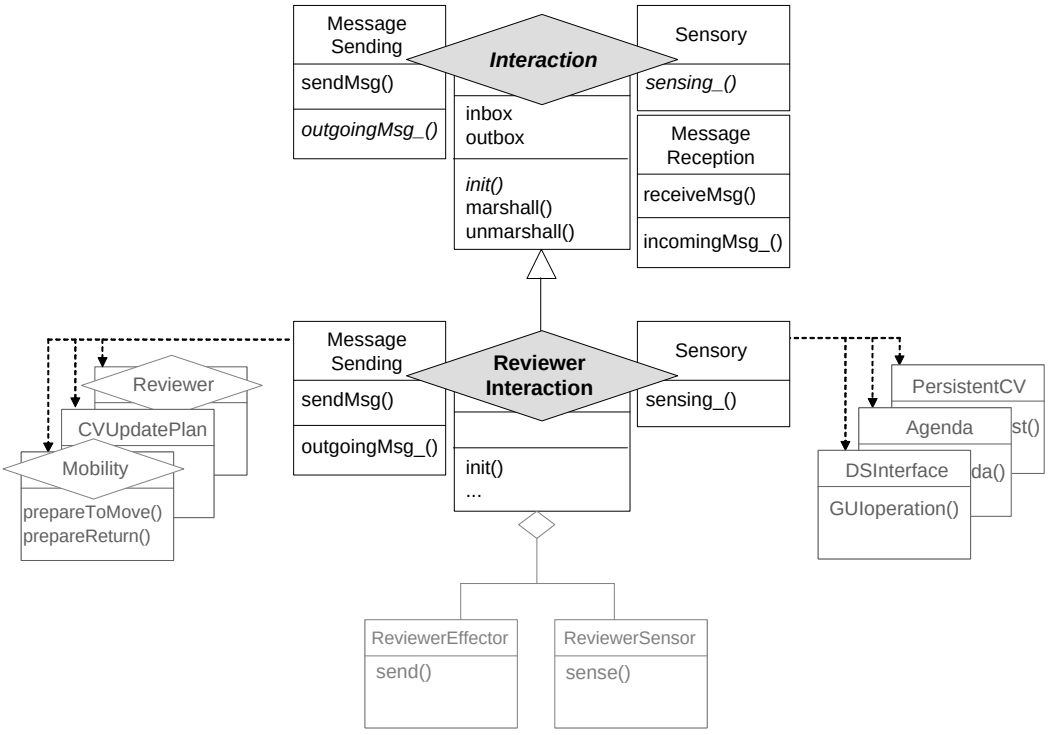


Figura 46. O padrão de interação para o agente de usuário do EC.

Por exemplo, o aspecto ReviewerInteraction afeta dois métodos do aspecto de Mobilidade (Seção 5.8) porque o agente revisor precisa enviar mensagens de notificação para agentes locais antes de mover para um ambiente remoto e depois de retornar ao ambiente original. Depois das execuções do método prepareToMove() (join point), o aspecto ReviewerInteraction intercepta o fluxo de controle do programa e o advice outgoingMsg_ envia uma mensagem aos outros agentes de ambiente, notificando-os de que esse agente está se deslocando para um ambiente remoto e não pode mais receber solicitações de serviço locais. Depois da execução do método prepareReturn() (join point), o aspecto ReviewerInteraction assume o controle da execução e o advice envia uma mensagem aos outros agentes de ambiente, notificando-os de que o agente revisor está de volta e já pode receber novas solicitações de serviço. A Figura 46 também descreve o relacionamento entre o aspecto ReviewerInteraction e a classe de plano CVUpdatePlan. Esse relacionamento existe porque uma mensagem precisa ser enviada a um agente quando o currículo do revisor é modificado.

Dinâmica. Os cenários a seguir descrevem o comportamento dinâmico do padrão Interação.

Cenário I – Recebendo uma mensagem de outro agente, ilustrado pela Figura 47a, apresenta o comportamento do padrão quando os aspectos de interação recebem mensagens de outros agentes em pontos específicos no fluxo de execução:

- O sensor recebe uma mensagem da plataforma associada.
- O sensor delega a desmontagem da mensagem e a tradução dos processos para as classes auxiliares.
- O aspecto de Interação detecta que uma mensagem foi recebida interceptando o método sense() (join point) da classe Sensor.
- Esse aspecto captura a mensagem recebida e atualiza a caixa de entrada.

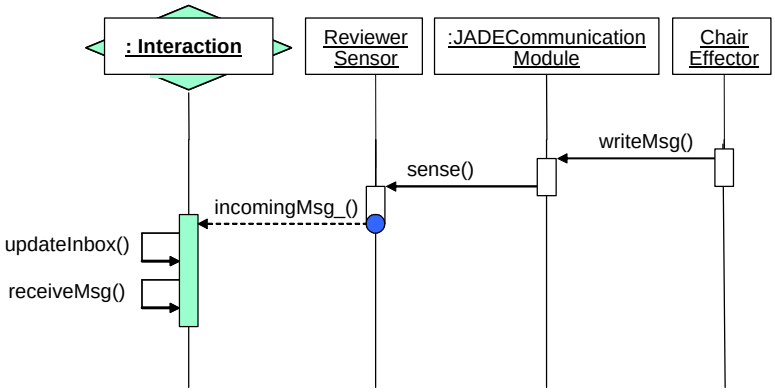


Figura 47a. Padrão Interação: recebendo uma mensagem.

Cenário II – Enviando uma mensagem, ilustrado pela Figura 47b, apresenta o comportamento do padrão quando os aspectos de interação detectam a necessidade de envio de mensagens em pontos específicos do fluxo de execução:

- O agente começa executando uma de suas ações ou planos.
- O aspecto Interaction detecta pontos no fluxo de execução (join points) das ações do agente ou planos em que uma mensagem precisa ser enviada.
- Esse aspecto monta a mensagem recebida e atualiza a caixa de saída do agente.
- O aspecto envia a mensagem selecionando o efetor apropriado.
- O efetor delega a tradução da mensagem para as classes auxiliares.
- O efetor envia a mensagem ao ambiente usando a plataforma associada.

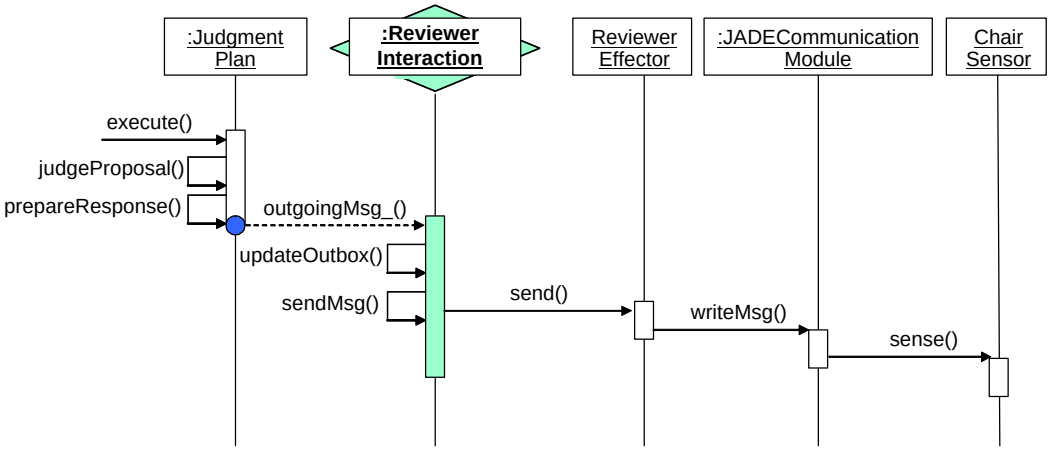


Figura 47b. Padrão Interação: enviando uma mensagem.

Cenário III – Detectando um evento externo em uma classe Environment, ilustrado pela Figura 48, apresenta o comportamento do padrão quando os aspectos de interação detectam estímulos das classes no ambiente:

- O aspecto de interação observa um evento a partir do ambiente interceptando alguma execução de método ou atualização de atributo em classes do ambiente.
- O aspecto Interaction seleciona um sensor de ambiente.
- O sensor do ambiente delega a desmontagem da mensagem e a tradução para as classes auxiliares.
- O aspecto Interaction detecta que uma mensagem foi recebida interceptando o método sense() (join point) do sensor.
- Esse aspecto captura a mensagem recebida e atualiza a caixa de entrada.

Observe que o comportamento sensorial e a recepção de mensagens são tratados de forma uniforme. A única diferença é que os aspectos precisam inspecionar diretamente as classes do ambiente a fim de detectar de forma transparente eventos externos, enquanto as mensagens são recebidas pelas classes Sensor, associadas a plataformas de comunicação.

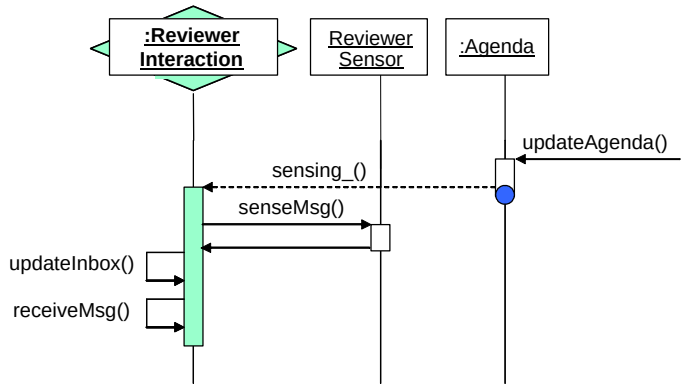


Figura 48. Padrão interação: detectando um evento externo.

Conseqüências. O uso do padrão Interação oferece as vantagens a seguir:

- *Melhor Separação de concerns*. O protocolo de interação é totalmente separado dos demais concerns do agente e das classes do ambiente. Os construtos orientados a aspectos oferecem suporte à definição separada do envio de mensagens e do comportamento sensorial que afeta muitas unidades

do sistema. Essa separação de concerns permite uma melhor modularidade, evitando o código entrelaçado e o espalhamento do mesmo em várias unidades. Portanto, a manutenibilidade e a reusabilidade também sofrem uma melhora.

- *Menor número de componentes e menos acoplamento.* O padrão Interação reduz o número de classes e inter-relacionamentos adicionais existentes na solução baseada em adaptadores [137]. Não é necessário criar outros adaptadores ou um plano de detecção para implementar o comportamento detecção.
- *Transparência.* O uso de aspectos é uma solução eficaz para a introdução de comportamento de detecção nas classes da aplicação não desenvolvidas para serem detectadas. As classes do ambiente são observadas de forma transparente.
- *Facilidade de evolução*. Com a evolução do sistema multiagentes, novos planos precisam enviar mensagens ou novos objetos externos podem ter de ser detectados pelos agentes. Os desenvolvedores de SMAs só precisam de novos pointcuts no aspecto Interaction a fim de implementar a nova funcionalidade necessária.
- *Suporte para a observação de componentes de terceiros.* Às vezes, as classes observadas podem fazer parte de componentes de terceiros, dos quais não temos acesso ao código-fonte. O padrão Interação ainda é aplicável uma vez que as linguagens orientadas a aspectos, como AspectJ, oferecem suporte ao processo de combinação de bytecode. Nesse caso, apenas os arquivos .class são necessários para a realização da combinação.

Todavia, essa solução de padrão possui as desvantagens a seguir:

- *Concern inseparável*. O padrão Interação não modulariza a montagem da mensagem a partir de diferentes planos ou papéis; a mensagem precisa ser preparada em classes de plano ou em aspectos de papel, porque sua montagem é muito acoplada ao contexto de planos ou papéis. Uma solução seria separar

a montagem da mensagem com aspectos, mas isso resultaria em uma maior complexidade.

- *Definições repetitivas e que consomem tempo.* Todos os emissores de mensagens do sistema precisam ser especificados no pointcut dentro do aspecto Interaction. Isso pode, de fato, ser repetitivo e tedioso, sugerindo que AspectJ deve ter construtos de metaprogramação mais poderosos. Contudo, esse problema não é insolúvel, porque as ferramentas de geração de código podem auxiliar os engenheiros de SMAs nessa etapa de desenvolvimento (Seção 5.11). Além disso, podemos estabelecer uma convenção de nomenclatura e usar wildcards suportados pela maioria das linguagens orientadas a aspectos [12] (Apêndice I). A implementação do Portalware e do Expert Committee usou convenções de nomenclatura.

Usos conhecidos. O Portalware e o Expert Committee implementaram o padrão Interação. A implementação de uma arquitetura simuladora de tráfego [51] também usou o padrão Interação (Capítulo 7).

Padrões relacionados. O padrão Interação está relacionado ao padrão Kernel do Agente, porque associa o comportamento interativo a diferentes classes desse padrão. Ele introduz a caixa de entrada e caixa de saída na classe Agent. Intercepta ações em classes do agente e planos a fim de enviar mensagens ao ambiente externo. O padrão Interação também está conectado ao padrão Mobilidade, conforme demonstrado anteriormente. O padrão Autonomia e o padrão Adaptação dependem do padrão Interação porque eles interceptam as recepções de mensagens. O comportamento autonomia inspeciona as mensagens para tomarem decisões e instanciam os objetivos a fim de reagir a solicitações externas e mudanças de ambientes. O comportamento adaptativo trata das mensagens para adaptar as crenças do agente. Finalmente, o padrão Papel está relacionado ao padrão Interação, uma vez que as ações colaborativas dentro dos papéis precisam enviar mensagens a outros agentes.

5.5
O padrão Autonomia

Intenção. O padrão Autonomia permite a transformação de um objeto em um agente autônomo. Ele auxilia os engenheiros de SMAs ao oferecer a definição de três dimensões de autonomia de forma transparente para o objeto passivo.

Também conhecido como. *Padrão Independência, Padrão Iniciativa*.

Contexto. Os agentes de software são autônomos (Seção 3.3.4) e, como conseqüência, têm suas threads de controle (autonomia de execução), toma decisões (autonomia de decisão) e executa ações e planos a seu critério (autonomia proativa). Desejamos transformar de forma transparente objetos passivos em agentes autônomos.

Exemplo de motivação. Os agentes de EC incorporam as três dimensões de autonomia. Em relação à autonomia de execução, cada agente é multisegmentado e possui estratégias distintas para instanciar suas threads: “thread por solicitação” e “pooling de threads”. Em relação à autonomia de decisão, os agentes de usuário precisam: (i) tomar decisões quando as mensagens são recebidas de outros agentes – por exemplo, quando o agente revisor recebe uma proposta de revisão de um agente chair e (ii) quando são recebidos estímulos externos dos objetos do ambiente – por exemplo, quando a agenda do usuário está cheia, e o agente propõe ao usuário deixar o comitê da conferência. Em relação à autonomia proativa, os agentes do usuário realizam atividades proativas em diferentes circunstâncias. Por exemplo, quando a agenda está cheia, o agente revisor sugere que o usuário deixe o comitê do programa. O projeto orientado a objetos do concern de autonomia normalmente afeta várias classes do agente.

Problema. O concern de autonomia não é bem capturado pelas abstrações orientadas a objetos. Em geral, parte do protocolo de autonomia normalmente afeta várias classes, incluindo as classes do agente, plano e papel. Como conseqüência, não é possível transformar objetos passivos existentes em entidades autônomas de forma que o objeto passivo seja transparente em relação ao comportamento autônomo

introduzido. Como melhorar a separação entre o comportamento autônomo e outros concerns do agente? As forças a seguir moldam a solução do padrão:

- A solução do projeto deve oferecer suporte aos desenvolvedores de SMAs para que façam objetos passivos em entidades autônomas de forma transparente para os objetos.
- A manutenção do comportamento autonomia não deve afetar a funcionalidade básica do agente.
- O solução do projeto deve expor claramente as relações existentes entre o concern de Autonomia e outros concerns do agente, como adaptação, interação etc.

Solução. Usar aspectos para melhorar a separação do comportamento autônomo. Os aspectos Autonomy são usados para modularizar o comportamento de autonomia que afeta algumas unidades da modularidade de um agente de software. Um aspecto Autonomy (Figura 49) separa o comportamento autônomo dos elementos do kernel do agente, como planos, crenças e de outros aspectos do agente, como o aspecto Interaction. Esse aspecto ajuda a capturar os eventos que disparam os objetivos proativos e os objetivos de decisão, que não são fáceis de modularizar com base apenas em abstrações orientadas a objetos. Esse aspecto captura a instanciação de threads que oferecem suporte à execução autônoma de um agente. Como o comportamento autônomo completo é definido em um único aspecto, a propriedade de autonomia pode ser acoplada e desacoplada de objetos passivos em uma forma *plug-and-play*. Ele gerencia threads do agente e agrega a definição de eventos quando o agente precisa tomar decisões e iniciar atividades internas sem solicitações externas.

Usando o aspecto Autonomy, os desenvolvedores de agente podem definir em um único módulo quando os planos de tomada de decisão e os planos proativos podem ser chamados. As chamadas a esses planos não são entrelaçadas com as classes do agente básicas, que fazem parte do kernel do agente. O aspecto Autonomy afeta os pontos de execução dessas classes e muda sua execução normal a fim de tomar decisões e instanciar objetivos reativos e proativos. Além do mais, o aspecto Autonomy pode ser usado para implementar o controle do grau de autonomia. A

solução é informar (ou seja definir pointcuts para) todos os métodos que influenciam a confiabilidade do agente.

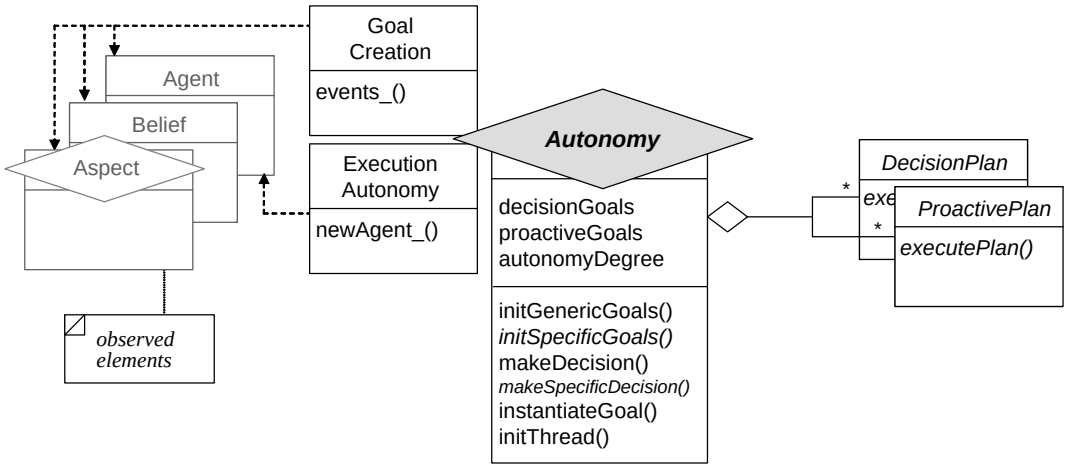


Figura 49. A visão estática do padrão Autonomia.

Estrutura. O padrão Autonomia possui um participante principal e dois tipos de clientes:

Participante Principal:

- **Aspecto Autonomy**

- define a lógica básica do concern de autonomia.

Participantes cliente:

- **Objeto passivo**

- é o objeto que precisa ficar autônomo.

- **Gerador de evento**

- gera eventos que provavelmente darão início a uma criação de objetivos.

A Figura 49 apresenta as partes comuns a todas as instanciações potenciais do padrão e outras partes que não são específicas a cada instanciação. As partes comuns são:

1. A inicialização de objetivos genéricos.

2. A especificação dos eventos genéricos que disparam os algoritmos de tomada de decisão e as atividades proativas.

3. O protocolo de autonomia geral:

a. uma thread é acoplada a um objeto que pretende ser autônomo.

b. os eventos são detectados, as decisões são tomadas e os objetivos são instanciados.

4. A interface para instanciar os objetivos.

As partes específicas para cada instanciação do padrão são:

1. A inicialização de objetivos específicos.

2. Um conjunto de eventos específicos que provavelmente dispararão os algoritmos de tomada de decisão e as atividades proativas.

O propósito do aspecto Autonomy é fornecer para as instâncias Agente autonomia de execução, autonomia de decisão e autonomia proativa. O aspecto Autonomy adiciona threads de controle às instâncias Agente. Esse aspecto também implementa a criação de objetivos proativos e reativos, que são precedidos por decisões quando apropriado. As subclasses DecisionPlan e ProactivePlan modularizam a implementação do comportamento proativo e algoritmos de decisão mais sofisticados. O aspecto Autonomy possui três partes principais: o próprio aspecto e duas interfaces crosscutting. O aspecto tem os objetivos proativos e de decisão, um número inteiro representando o grau de autonomia, os métodos de inicialização e define o protocolo de autonomia. O aspecto Autonomy é abstrato e precisa ser estendido para implementar o comportamento autônomo para contextos específicos de tipos de agente e papéis.

As interfaces crosscutting definem como o aspecto Autonomy afeta diferentes classes de agentes de software. A interface GoalCreation define os geradores de evento. Essa interface especifica join points nas classes do agente, nas quais os eventos precisam ser detectados para dar início à criação de um objetivo. Ela contém um advice que executa depois de execuções de ações em classes do agente, ações em classes de crença, ação em classes de plano e ações em outros aspectos associados a agentes (por exemplo, os aspectos de interação – Seção 5.4). Como o aspecto

Autonomy implementa o protocolo de autonomia, ele é associado às classes do agente, plano ou crença nas quais as alterações a seu estado podem disparar a criação de um objetivo. Atividades de tomada de decisão específicas (classes DecisionPlan) e atividades proativas (classes ProactivePlan) são implementadas em classes separadas e seu isolamento do aspecto Autonomy é desejável para melhorar a reutilização. A interface ExecutionAutonomy especifica quando e como uma instância de thread será associada a instâncias de agente. Essa interface especifica o construtor de agente como join point a partir do qual a thread deve ser instanciada. Ela contém um advice que executa depois das execuções de construtores da classe Agent.

A Figura 50 ilustra a instanciação de padrões do papel de revisor no sistema Expert Committee. O aspecto Autonomy e seus subaspectos afetam classes e outros aspectos do agente nesse sistema. Ele mostra a classe Agent, a classe Belief, o aspecto Interaction, a classe Agenda e o aspecto Reviewer. Há outros, mas eles seguem essencialmente o mesmo padrão. O aspecto Autonomy afeta o método receiveMsg() do aspecto Interaction (Seção 5.4), porque um agente normalmente precisa decidir se e qual instância de objetivo reativo deve ser criada dependendo das mensagens recebidas. Depois da execução do método receiveMsg() (join point), o aspecto Autonomy assume o controle da execução do programa e instancia, se necessário, as decisões de objetivo associadas ao tipo de mensagem. Se a decisão for positiva, é instanciado um objetivo reativo. Caso contrário, um plano de decisão enviará uma mensagem ao emissor notificando o agente de que a solicitação de serviço não será realizada, e o objetivo reativo não será instanciado.

A Figura 50 também descreve o relacionamento entre o aspecto Autonomy e a classe Agent. Esse relacionamento existe porque uma instância Thread (padrão Objeto Ativo [153]) precisa ser associada a cada nova instância Agente, ou seja, o aspecto introduz a capacidade de autonomia no objeto Agente passivo. A interface ExecutionAutonomy especifica o construtor do Agente como um join point a partir do qual a thread será ativada.

ReviewerAutonomy é definido como um subaspecto de Autonomia. Ele especializa a interface crosscutting GoalCreation a fim de definir os eventos ou join points em que são instanciados objetivos proativos e de decisão. Também implementa

os métodos abstratos initSpecificGoals() e makeSpecificDecision(). Os planos proativos e a decisão específica estão associados a esse subaspecto. O aspecto Autonomy possui outros subaspectos para o papel de chair e para o agente do usuário, mas eles seguem, essencialmente, a mesma estrutura de ReviewerAutonomy.

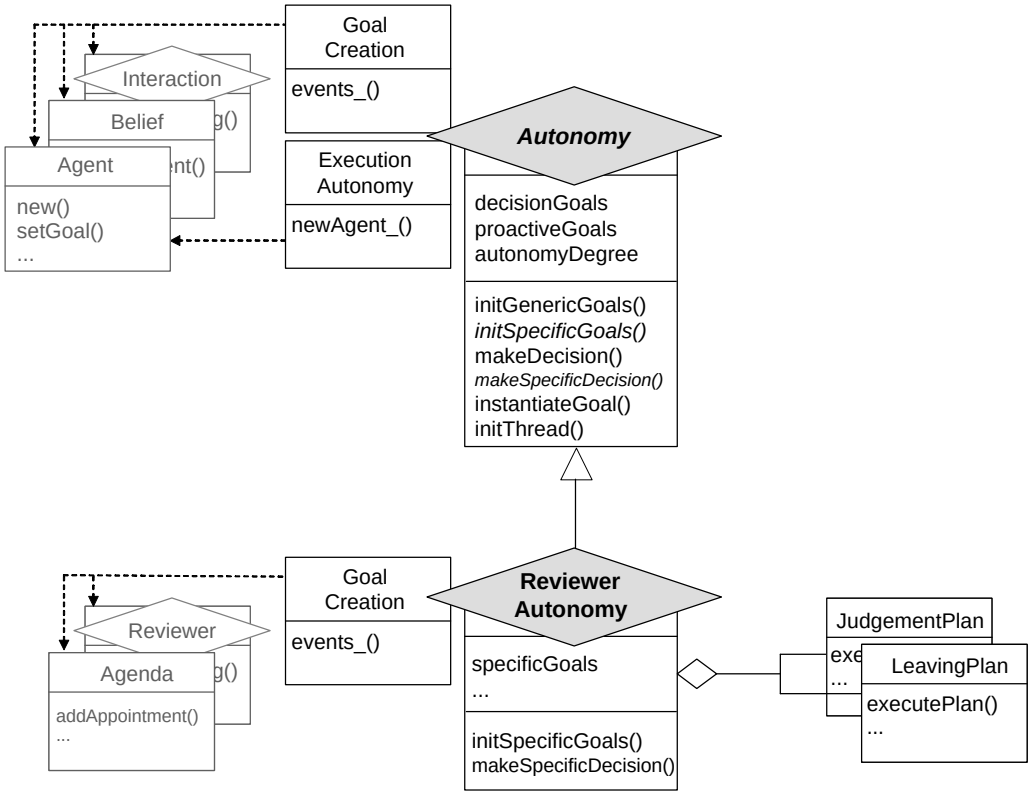


Figura 50. O padrão Autonomia do papel de Revisor.

Dinâmica. Os cenários a seguir descrevem o comportamento dinâmico do padrão Autonomia.

Cenário I – Tornando um agente autônomo, ilustrado na Figura 51, apresenta o comportamento do padrão quando o aspecto de autonomia associa o agente a uma thread de controle e cria um objetivo reativo no recebimento de uma mensagem externa:

- O agente é criado pela chamada desse construtor.

- O aspecto Autonomy informa sobre a criação de agente e fornece execução de autonomia à instância de agente – ou seja ela incorpora uma thread de controle a essa instância.
- Sempre que uma mensagem é recebida, o aspecto Autonomy tenta encontrar objetivos de decisão associados ao evento específico (ou seja, o tipo de mensagem recebida).
- O aspecto cria os objetivos de decisão, que podem ser genéricos ou específicos a uma função ou tipo de agente, e executa os planos de decisão associados.
- É definido um novo objetivo.

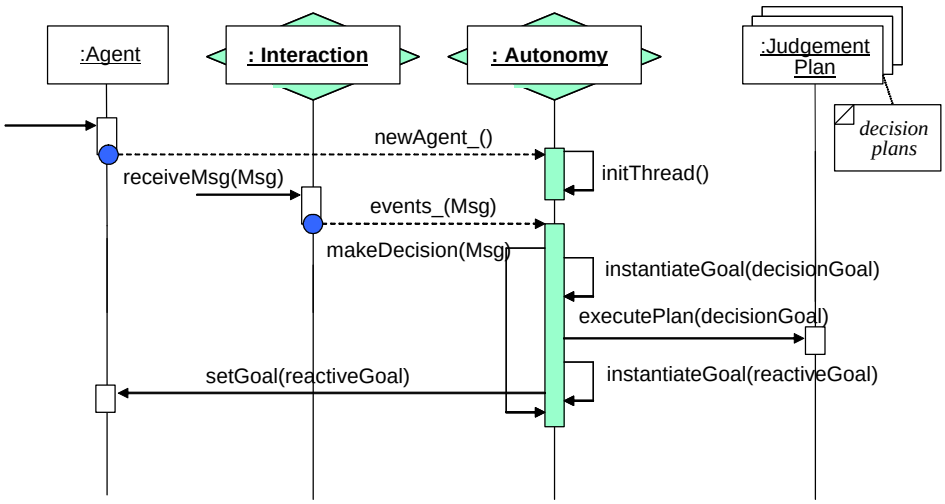


Figura 51. Padrão Autonomia: tornando um agente autônomo.

Cenário II – Instanciando um objetivo proativo, ilustrado na Figura 52, apresenta o comportamento do padrão quando o aspecto Autonomy detecta um evento que é um candidato a disparar um comportamento proativo:

- O aspecto de autonomia detecta o estabelecimento de um novo compromisso de usuário (eventos interno) que pode ser uma razão para o início do objetivo proativo de propor ao usuário que deixe o Comitê do Programa do qual está participando.
- O objetivo será definido somente se o grau de autonomia do agente for maior do que zero.

- O aspecto Autonomy tenta encontrar objetivos de decisão associados a esse evento específico.
- O aspecto cria os objetivos de decisão, que podem ser genéricos ou específicos a um papel ou tipo de agente, e executa os planos de decisão associados.
- É definido o novo objetivo.

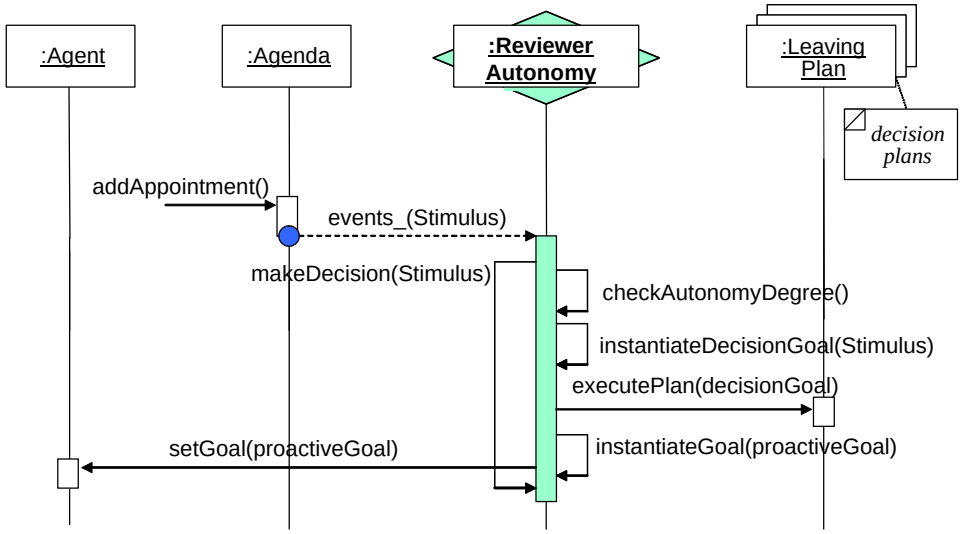


Figura 52. Padrão Autonomia: Criando um objetivo proativo.

Conseqüências. A solução de padrão fornece os benefícios a seguir:

- *Melhor Separação de concerns*. O concern de autonomia é modularizado nos aspectos Autonomy. Os objetos não implementam comportamento autônomo e não são alterados a fim de implementar o protocolo de autonomia.
- *Flexibilidade.* São muitas as dimensões da autonomia exploradas nas diferentes aplicações de SMAs. Os agentes autônomos simples só têm autonomia de execução (objetos ativos), enquanto os tipos de agente sofisticados também incorporam comportamento proativo. Como o padrão desacopla o comportamento autônomo em aspectos, as implementações específicas da autonomia de agente são facilmente configuradas pelo desenvolvedor de agentes.

- *Configuração dinâmica.* A estratégia da thread pode precisar ser controlada dinamicamente de acordo com a carga de trabalho do agente. A implementação da configuração dinâmica da estratégia de threads é transparente para os outros módulos do agente, uma vez que pointcuts podem ser usados para detectar o grau de atividade do agente e alterar a estratégia em tempo de execução.
- *Melhor suporte para manutenção*. Há duas questões principais no protocolo de autonomia que normalmente alteram: (i) os pontos de execução de agente em que as threads de agente são iniciadas e (ii) os eventos que disparam alguma decisão de agente. Como a definição desses pontos e eventos está localizada em um único módulo, o aspecto Autonomy , a manutenibilidade do agente é melhorada.

Por outro lado, o uso do padrão Autonomia possui algumas desvantagens:

- *Refatoração necessária*. Em algumas circunstâncias, a realização do padrão Autonomy requer a reestruturação do código-base associado aos outros componentes do agente a fim de expor join points adequados. Por exemplo, precisamos garantir que cada método que pede a confirmação do usuário (quando uma decisão de agente é tomada) retorne um valor booleano. Isso permite que o aspecto capture a resposta do usuário e controle o grau de autonomia do agente. Ademais, extraímos o código dos métodos existentes para um novo método para expor um join point no nível do método. As ferramentas para ajudar a reestruturação facilitariam a introdução de aspectos em um sistema existente. (Seção 5.11.2).
- *Estrutura complexa para agentes simples*. Alguns agentes reativos simples não precisam de controle de threads, eles só reagem a alguns eventos, tomam decisões muito simples e não têm comportamento proativo. Nesse caso, o código da autonomia tende a ser localizado em menos métodos. O uso de aspectos nessa situação específica pode aumentar em vez de diminuir a complexidade do agente.

Variantes. *Autonomia reativa*. Esse variante não contém a especificação dos eventos que dispariam os objetivos proativos. É apropriado para agentes que não sejam proativos, mas são multisegmentados e tomam decisões com base em eventos. Esse variante é similar ao padrão Autonomia. No entanto, a interface crosscutting GoalCreation é mais simples uma vez que escolhe apenas as recepções de mensagens.

Autonomia reflexiva. Esse variante implementa uma versão reflexiva do padrão Autonomia. Ele separa o comportamento autonomia de outros concerns do agente com base em metaobjetos, em vez de em objetos. Um protocolo metaobjeto intercepta dinamicamente os eventos e redireciona o fluxo de controle para os metaobjetos que implementam o comportamento autonomia. A desvantagem desse variante é que não é fácil compor os metaobjetos de autonomia com metaobjetos implementando outros concerns de agente.

Usos conhecidos. Os agentes de EC usam o padrão Autonomia, conforme descrito aqui. Os agentes do Portalware implementam o variante *Autonomia reativa* do padrão, porque não são proativos. Briot et al. [112] e Amandi [7, 8] implementaram suas arquiteturas com base no variante reflexivo do padrão Autonomia.

Padrões relacionados. O padrão Autonomia está relacionado ao padrão Interação porque precisa informar sobre os recebimentos de mensagens. Esse padrão depende do padrão Kernel do Agente porque introduz o comportamento autonomia nas classes de conhecimento, como a classe Agent. O padrão Autonomia pode cooperar com o padrão Aprendizagem – quando um aspecto Aprendizado infere novas conclusões, ele pode disparar um comportamento de agente proativo. Um aspecto Autonomy pode interceptar as inferências de aprendizagem. Finalmente, a estrutura de autonomia está relacionada ao padrão Objeto Ativo [153] porque oferece suporte à definição flexível da estratégia de controle de threads. Esse padrão foi usado no sistema do EC para implementar duas estratégias distintas para a instanciação das threads de agente: “thread por solicitação” e “pooling de threads”.

5.6
O padrão Adaptação

Intenção. O Padrão Adaptação abstrai o comportamento adaptativo do agente. Ele modulariza o protocolo para a adaptação de comportamento e conhecimento, desacoplando a funcionalidade principal do agente dos componentes de adaptação. A solução do padrão permite que os programadores de agente conectem as estratégias de adaptação às classes Agent Kernel de forma transparente.

Contexto. Os agentes precisam ser capazes de adaptar seu conhecimento e comportamento à natureza dinâmica dos ambientes nos quais operam. O protocolo de adaptação consiste em observar os eventos relevantes, juntando as informações necessárias, selecionando e chamando os adaptadores associados. O comportamento e a estrutura do agente podem ter de ser adaptados em várias circunstâncias: quando as crenças são alteradas, quando um novo objetivo foi alcançado etc (Seção 3.3.3). O agente precisa ser capaz de observar sua estrutura e comportamento a fim de adaptar o conhecimento e o comportamento. Desejamos descrever o protocolo de adaptação o mais separadamente possível.

Exemplo de motivação. No sistema EC, a adaptação é necessária em várias circunstâncias. a chegada de uma nova mensagem, a definição de um novo objetivo, a falha de um plano, as alterações de crença etc. Esses eventos são eventos importantes que precisam ser observados e provavelmente dispararão algum tipo de adaptação. Quando chega uma proposta de revisão do agente chair, o agente revisor precisa adaptar suas crenças, como o prazo final para o envio de uma resposta à chair. A alteração de crenças pode, por sua vez, resultar na alteração de crenças associadas. Quando é definido um novo objetivo, deve ser selecionado um novo plano.

Kendall propõe a implementação do comportamento adaptativo usando o padrão Observador [137]. Essa solução trabalha bem porque a intenção do padrão Observador é “definir uma dependência um-para-muitos entre objetos de forma que quando um objeto muda de estado, todos os dependentes são notificados e atualizados automaticamente” [79]. As implementações orientadas a objetos do padrão Observador normalmente adicionam uma referência a todos os assuntos potenciais (as

classes observadas) que armazenam uma lista de observadores interessados em alterações desse assunto particular. Quando um assunto deseja reportar um estado para seus observadores, ele chama seu próprio método notify(), que, por sua vez, chama uma método de atualização para todos os observadores na lista. Como conseqüência, essa solução possui algumas desvantagens: (i) ela diminui a separação entre os concerns básicos do agente e o concern de adaptação, (ii) ela abusa do mecanismo de herança (3.6.2) uma vez que todas as crenças observadas precisam implementar a interface Observável, aumentando a profundidade geral da herança nos SMAs e (iii) as classes kernel (Seção 5.3) são muito acopladas a componentes de adaptação, dificultando a evolução do sistema.

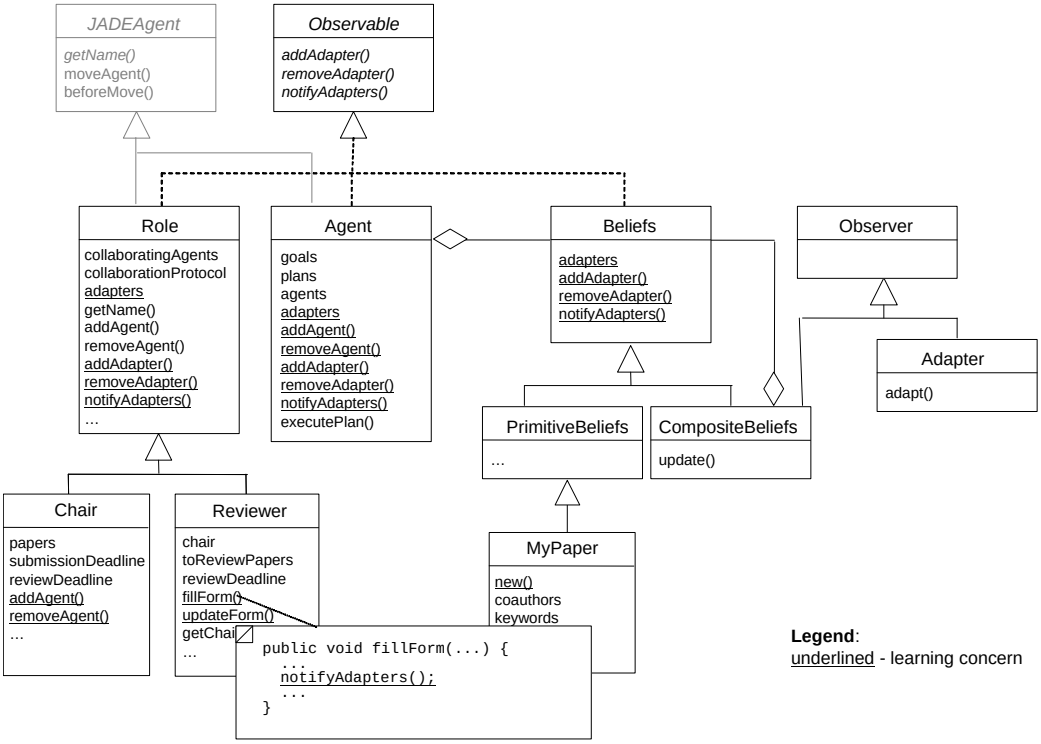


Figura 53. Comportamento adaptação nos agentes do EC. O padrão Observador [137].

Considere um exemplo concreto do padrão Observador implementado na linguagem Java no contexto da adaptação de conhecimento, conforme demonstrado na Figura 53. Nessa solução, o padrão Observador seria usado para causar operações mutantes em elementos de crença a fim de notificar os componentes de adaptação. Conforme mostrado na figura, o código para a implementação do padrão Observador

é espalhado pelas classes belief . O componente Observável é uma interface e uma classe não-abstrata porque as classes observáveis já estendem uma classe abstrata (JADEAgent). Java, a linguagem orientada a objetos mais comum e expressiva para a implementação de SMAs, não oferece suporte a várias heranças. Todos os participantes (por exemplo Chair) estão interligados por métodos do padrão Observador e as respectivas chamadas de métodos e, conseqüentemente, possuem código de adaptação. A adição ou remoção do concern de adaptação de uma classe requer alterações invasivas nessa classe. A alteração do protocolo de adaptação requer alterações em todas as classes participantes.

Problema. Os eventos e as estratégias associados à adaptação do agente variam muito. Em geral, o protocolo de adaptação afeta as diversas classes associadas ao kernel do agente. A adaptação de conhecimento afeta muitos elementos, incluindo os papéis, as classes do agente e as classes da crença. A adaptação do comportamento afeta diferentes elementos, como classes do agente e planos. Seria interessante se o kernel do agente e outros concerns pudessem ser decompostos a partir do protocolo de adaptação. Como melhorar a separação entre o comportamento adaptativo e outros concerns do agente? As forças a seguir surgem ao tratar desse problema:

- A alteração dos eventos e estratégias não deve afetar a funcionalidade básica do agente.
- Deve ser fácil reutilizar os protocolos básicos da adaptação de conhecimento e adaptação de comportamento por diferentes papéis e tipos de agente.

Solução. Usar aspectos para melhorar a separação do comportamento adaptativo do agente. Os aspectos Adaptation são usados para capturar o protocolo de adaptação que afeta muitas partes de um agente de software. Um aspecto Adaptation (Figura 54) separa o comportamento adaptativo dos elementos do kernel do agente, como planos, papéis, crenças etc. Em outras palavras, os aspectos são usados não só para modularizar o centro do comportamento adaptativo, mas também para isolar todo o comportamento relacionado ao protocolo de adaptação, incluindo a adaptação de comportamento e conhecimento.

Usando os aspectos Adaptation, definimos quando e como o agente é adaptado. Eles conectam os pontos de execução do programa (eventos) aos elementos do agente que precisam ser adaptados às estratégias de adaptação correspondentes. Esses aspectos conseguem afetar alguns pontos de execução do agente – por exemplo chamadas a métodos nas classes Agent – e alteram sua execução normal a fim de disparar um adaptador ou componente de adaptação. Eles monitoram esses pontos de execução a fim de identificar quando uma adaptação deve ser disparada. Em geral, quando um determinado elemento do conhecimento é alterado, essa alteração é candidata a motivar uma adaptação. As classes auxiliares são usadas para implementar diferentes adaptadores, ou seja, diferentes estratégias de adaptação.

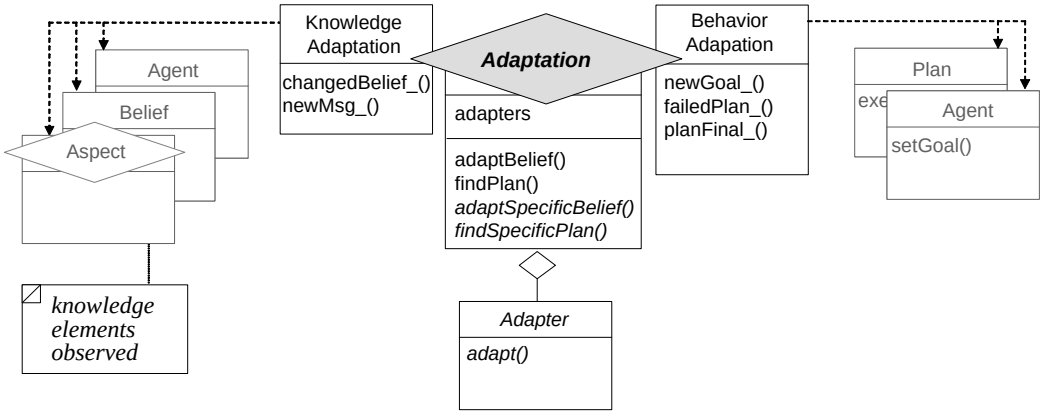


Figura 54. A visão estática do padrão Adaptação.

Estrutura. O padrão Adaptação possui dois participantes principais e um participante cliente:

Principais Participantes:

- **Aspecto Adaptation**

- define o protocolo de adaptação.

- **Adaptador**

- Implementa uma estratégia de adaptação específica.

Participante cliente:

- **Elemento do conhecimento**

- Oferece o contexto em que uma adaptação de comportamento ou um conhecimento pode ser disparado – pode ser uma classe da crença, uma classe do plano, uma classe do agente ou um aspecto do agente.

Na estrutura do padrão Adaptação (Figura 54), algumas partes são comuns a todas as instanciações potenciais do padrão e outras partes que são específicas a cada instanciação. As partes comuns são:

1. Os eventos genéricos em que o agente sempre precisa ser adaptado, por exemplo, quando é definido um novo objetivo ou quando ocorre uma exceção durante a execução de um plano.
2. O protocolo de adaptação geral: - os eventos são detectados, as condições são verificadas e as ações/planos disparados.
3. As adaptações genéricas.
4. A lista de adaptadores, ou seja, as referências a componentes de adaptação mais sofisticados (técnicas de busca, mecanismo de inferência e planejadores).

As partes específicas são:

5. Os eventos específicos associados a um determinado contexto (tipo de agente ou papel) em que o agente precisa ser adaptado – no qual as crenças precisam ser observadas de forma que seja necessária alguma adaptação.
6. As adaptações específicas.

O propósito do aspecto Adaptation é tornar as instâncias Agent adaptativas. O aspecto Adaptation estende o comportamento da classe Agent para introduzir o protocolo de adaptação. Esse aspecto contém os pointcuts que descrevem os eventos relevantes e as informações que precisam ser reunidas dos elementos do conhecimento. Ele contém os advices que chamam os métodos responsáveis pela implementação da adaptação ou de um adaptador específico. O aspecto Adaptation possui três partes principais: o próprio aspecto e duas interfaces crosscutting. O aspecto possui os métodos do adaptador e a lista dos adaptadores especializados. O

aspecto Adaptation é abstrato e precisa ser estendido para implementar o comportamento adaptativo para contextos específicos de tipos de agente e papéis.

As interfaces crosscutting definem como o aspecto Adaptation afeta as diferentes classes do sistema multiagentes. A interface KnowledgeAdaptation define os join points nos elementos do conhecimento que devem disparar uma adaptação de alguma parte do conhecimento. Ela descreve dois pointcuts principais: o pointcut changedBelief captura as alterações do conhecimento e o pointcut newMsg detecta a chegada de uma nova mensagem. Contém um advice que executa depois de execuções de ações de classes do agente, ações de classes do plano e outros aspectos associados ao agente (por exemplo, os aspectos de papéis – Seção 5.7). A interface BehaviorAdaptation define os join points nos elementos do conhecimento que devem disparar o cancelamento da execução de um plano ou a seleção de um novo plano. Como o aspecto Adaptation implementa o protocolo de adaptação, os aspectos da adaptação são associados a diferentes elementos do conhecimento, como as classes Agent, classes Plan, classes Belief e aspectos de papéis.

A Figura 55 ilustra a instanciação de padrões do papel de revisor no sistema Expert Committee. O aspecto Adaptation afeta diferentes classes e aspectos do agente nesse sistema (cerca de 9 componentes). No entanto, a figura somente apresenta um conjunto parcial das classes afetadas pelos aspectos. Ela mostra o aspecto Interaction, a classe Belief, a classe Agent, a classe Plan, a classe MyPaper e o aspecto Reviewer; os outros seguem, essencialmente, o mesmo padrão. Por exemplo, a adaptação do conhecimento é necessária quando é recebida uma mensagem do agente. Esse evento é detectado pela interceptação do método receiveMsg() no aspecto Interaction. A mensagem é capturada pelo pointcut newMsg na interface KnowledgeAdaptation, e a adaptação é realizada por meio de métodos internos do aspecto Adaptation.

A Figura 55 também ilustra um exemplo da adaptação do comportamento. A interface BehaviorAdaptation captura as alterações na lista de objetivos por meio da especificação do pointcut newGoal. Um after advice é associado ao pointcut e é responsável pela seleção de novos planos para alcançar o novo objetivo. Nesse caso, o método setGoal() é interceptado, o objeto do objetivo é capturado e um plano associado é chamado pelo advice.

O aspecto ReviewerAdaptation estende Adaptation para implementar o comportamento adaptativo específico ao papel de revisor. Ele refina a interface KnowledgeAdaptation para definir as crenças do revisor que disparam a adaptação do conhecimento. Por exemplo, o pointcut changedBelief afeta o construtor da classe MyPaper porque, como um novo trabalho é criado, há a necessidade de adaptar a crença que representa os interesses de pesquisa do revisor. As palavras-chave associadas ao novo trabalho precisam ser reunidas, e o método de adaptação dos interesses de pesquisa é chamado. Como alternativa, o adaptador de encadeamento é chamado a fim de inferir sobre novas áreas em que o usuário está interessado.

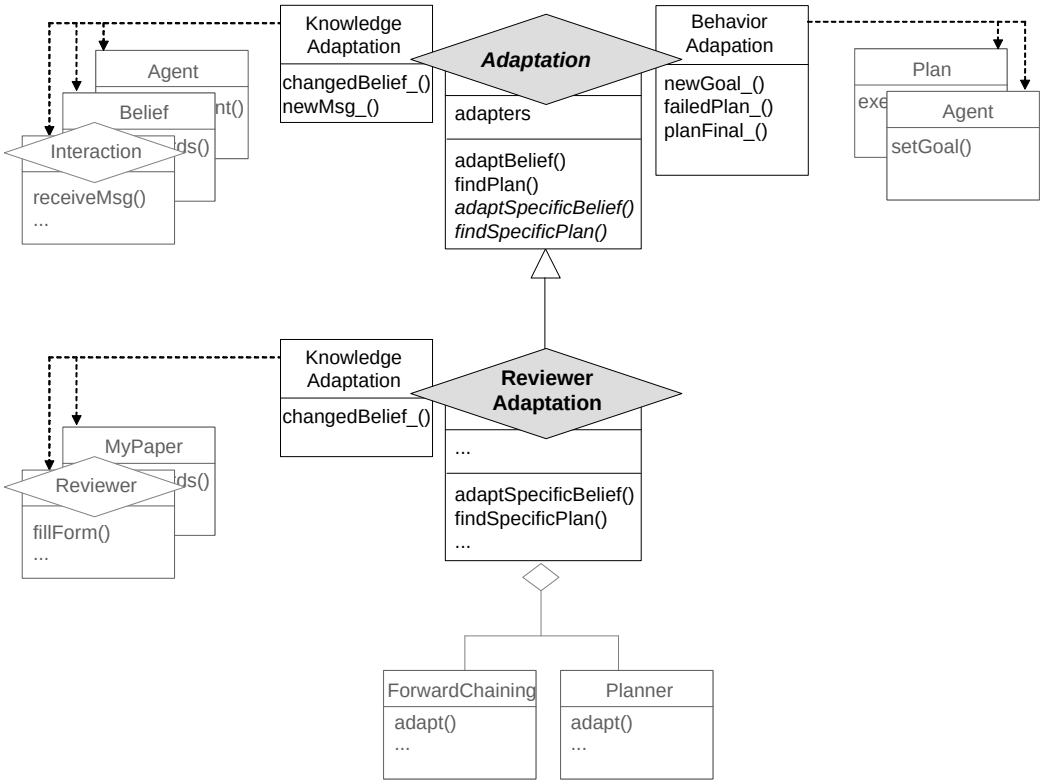


Figura 55. O padrão Adaptação do papel de revisor.

Dinâmica. Os cenários a seguir descrevem o comportamento dinâmico do padrão Adaptação.

Cenário II – Adaptando crenças de acordo com uma mensagem recebida, ilustrado pela Figura 56, apresenta o comportamento do padrão quando o aspecto

Adaptation detecta a necessidade de adaptação de crenças devido ao recebimento de mensagens de ambientes externos:

- O agente recebe uma mensagem do ambiente.
- O aspecto Adaptation detecta o recebimento de mensagens interceptando a operação receiveMsg().
- Esse aspecto reúne as informações necessárias dos elementos do conhecimento.
- O aspecto adapta diretamente as crenças do agente, conforme mostrado na Figura 56, ou seleciona um adaptador.

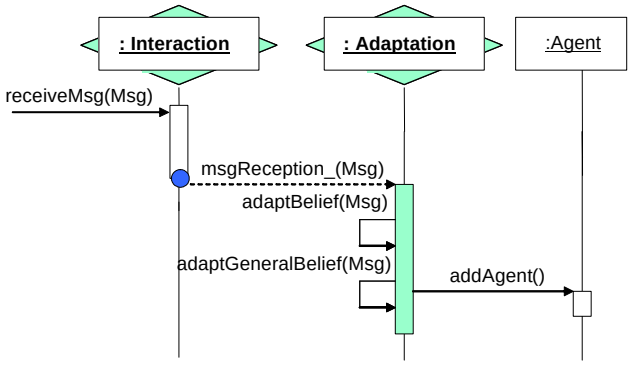


Figura 56. Padrão Adaptação: adaptando Agent no recebimento de uma mensagem.

Cenário II – Adaptando planos devido à definição de um novo objetivo, ilustrado pela Figura 57, apresenta o comportamento do padrão quando o aspecto Adaptation detecta um novo objetivo que deve ser alcançado:

- O agente possui um novo objetivo.
- O aspecto Adaptation detecta o novo objetivo interceptando a operação setGoal().
- Esse aspecto reúne as informações necessárias dos elementos do conhecimento (o objeto do objetivo nesse caso).
- O aspecto tenta encontrar um novo plano para o mesmo objetivo.
- O aspecto adiciona o objeto do objetivo à instância do novo plano.
- O aspecto adapta a lista de planos do agente que devem ser executados.
- O aspecto remove o plano dessa lista quando a execução do plano é concluída.

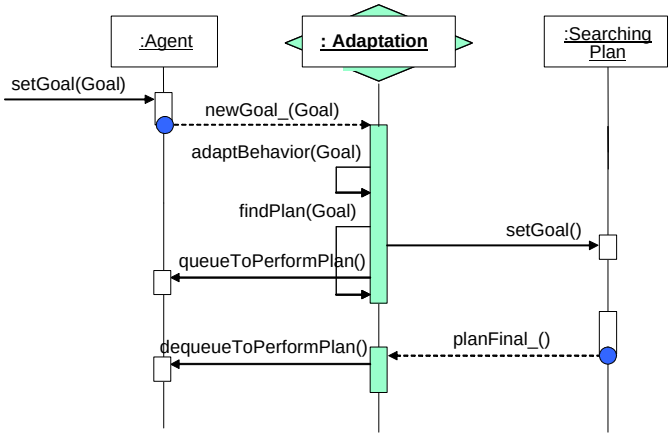


Figura 57. Padrão Adaptação: adaptando planos ao definir um novo objetivo.

Cenário III – Adaptando planos devido ao surgimento de uma exceção, ilustrado pela Figura 58, apresenta o comportamento do padrão quando o aspecto Adaptation detecta uma exceção em uma execução de plano:

- Uma exceção surge durante a execução de um método.
- O aspecto Adaptation detecta a exceção interceptando a resposta da operação executePlan().
- Esse aspecto reúne as informações necessárias, o objeto do plano nesse caso.
- O aspecto tenta encontrar um novo plano para alcançar o objetivo que o plano anterior não conseguiu alcançar.
- O aspecto adiciona o objeto do objetivo à instância do plano correspondente.
- O aspecto adapta a lista de planos do agente que devem ser executados.

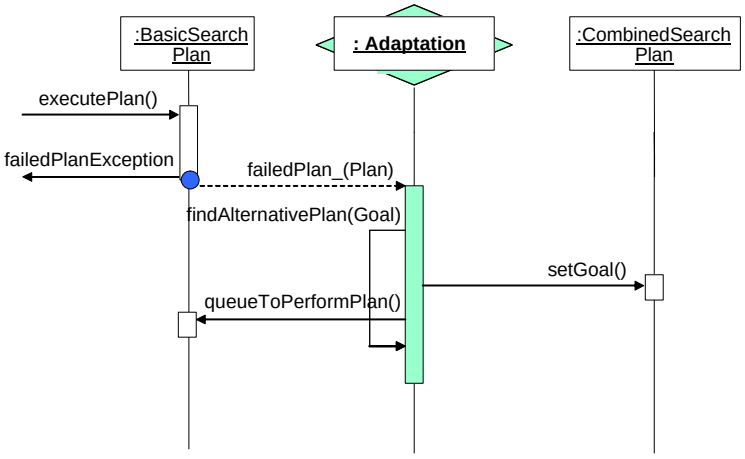


Figura 58. Padrão Adaptação: adaptando planos quando surge uma exceção.

Conseqüências. O padrão Adaptação possui as seguintes conseqüências:

- *Uniformidade*. O protocolo para a adaptação do conhecimento e a adaptação do comportamento é definido uniformemente no aspecto Adaptation. O código comum aos protocolos é reutilizado.
- *Melhor Separação de concerns*. O protocolo de adaptação é totalmente separado dos demais concerns do agente, como conhecimento e interação. As classes associadas aos demais concerns não possuem código de adaptação.
- *Menor replicação do código.* Conforme discutido anteriormente, o uso do padrão Observador [137] introduz a replicação do código à medida que a complexidade do agente aumenta. O padrão Adaptação oferece suporte à modularização desse código no aspecto Adaptation, minimizando o entrelaçamento e o espalhamento do código.
- *Transparência.* Os aspectos tornam-se uma abordagem elegante e poderosa que pode ser usada para introduzir o comportamento adaptativo em classes do agente de forma transparente. A descrição de quais classes do agente precisam ser observadas está presente no aspecto, e essas classes monitoradas não são modificadas de forma intrusiva.
- *Facilidade de evolução*. Com a evolução do sistema multiagentes, novas classes do agente precisam ser monitoradas e adaptadas. Os desenvolvedores de SMAs só precisam adicionar novos pointcuts e advices no aspecto Adaptation e conectá-los a adaptadores específicos a fim de implementar a nova funcionalidade necessária.
- *Fácil incorporação de adaptadores simples*. Muitas estratégias de adaptação são simples e não precisam de adaptadores sofisticados. Estratégias simples podem ser definidas como métodos em aspectos de adaptação.

Usos conhecidos. O Portalware e o Expert Committee implementaram o padrão Adaptação. Um sistema de gerenciamento de tráfego também usou o padrão Adaptação (Seção 7.1).

Padrões relacionados. O padrão Adaptação contém a implementação orientada a aspectos do padrão Observador [115]. O padrão Estratégia pode ser usado para implementar diferentes estratégias de adaptação [137]. O padrão Façade pode ser usado para fornecer uma interface única para as estratégias de adaptação, independente das técnicas de raciocínio ou algoritmos de planejamento. Finalmente, a implementação do padrão Adaptação (vide a seguir) usa alguns idiomas AspectJ [114], como *Template Advice*, *Composite Pointcut* e *Advice Method*.

5.7
O padrão Papel

Intenção. O padrão Papel oferece suporte à definição separada de papéis do agente. Ele decompõe as dificuldades do comportamento colaborativo da funcionalidade básica do agente.

Também conhecido como. *Papel colaborativo***.**

Contexto. Um agente exerce diferentes papéis em um sistema multiagentes (Seção 3.4.2). Os papéis capturam como os agentes interagem entre si em colaborações [136]. Um agente exerce papéis adicionais, que não fazem parte de sua funcionalidade básica. Cada papel envolve conhecimento extrínseco (Seção 3.4.2), ele possui suas próprias crenças, objetivos e planos. O papel também pode manipular o conhecimento intrínseco associado ao papel básico do agente. Com o aumento do número de papéis do agente, há a necessidade de modularizá-los. Precisamos fazer isso o mais separadamente possível.

Exemplo de motivação. Os agentes do usuário no sistema EC exercem os papéis atribuídos a eles. Esses agentes apresentam quatro papéis: chair, revisor, revisor adicional (revisor especial) e chamador (Seção 5.1). Um mesmo agente do usuário pode exercer vários papéis. O papel de chair possui objetivos específicos, incluindo o objetivo de distribuição de trabalhos a revisores (DistributionGoal) e o objetivo do contato de novos revisores quando o número de revisores não é suficiente (ContactGoal). Ele também possui vários planos para alcançar os objetivos e possui crenças que mantêm as informações da conferência, incluindo prazos finais, a lista de revisores e a lista de trabalhos enviados. O revisor possui crenças, objetivos e planos

específicos. O revisor adicional é convidado por um revisor para ajudar na revisão de um trabalho específico. O papel de chamador é associado ao papel de chair. É responsável pela colaboração com agentes de informação sempre que precisa do perfil de um dado revisor. Muitas abordagens foram usadas para a modularização de papéis em linguagens orientadas a objetos [19, 77, 108, 146, 241]. Em [77], Fowler avalia as diferentes abordagens. A mais comum é o padrão *Objeto do Papel* [19]. Esse padrão fornece uma classe individual a cada papel do agente. Cada classe do papel exibe conhecimento extrínseco e comportamento não-central, específicos ao contexto do papel. Os papéis são organizados em uma hierarquia, com subclasses para comportamento de papel mais especializado (AdditionalReviewer).

Kristensen e Osterbye [146] propõem o Objeto do Papel com o padrão Decorador [79] como o melhor suporte para os papéis em linguagens orientadas a objetos. A intenção do padrão Decorador é conectar dinamicamente responsabilidades adicionais a um objeto. No contexto dos papéis, os decoradores oferecem uma alternativa flexível para subclasses para estender os agentes com funcionalidades de papel. No contexto do padrão Objeto do Papel, a estrutura do Decorador é usada para o projeto da interface dos papéis.

A Figura 59 apresenta essa solução para os agentes do usuário do EC, sendo que uma linha descreve um caminho de colaboração entre dois papéis. A figura mostra que UserAgentRole e UserAgentType implementam a mesma interface CollaborativeUserRole (padrão Decorador). Um objeto que está usando uma instância de UserAgentType somente tem conhecimento de um objeto; no entanto, em tempo de execução, os papéis adicionam comportamento de forma transparente. Um objeto central (uma instância da classe UserAgentType) contém os papéis que exerce como um conjunto de instâncias do papel. A atribuição do papel dinâmico (conexão do papel) tem suporte porque as instâncias que representam os papéis atuais podem ser alteradas em tempo de execução.

Esse padrão permite o acoplamento e o desacoplamento dos objetos do papel do comportamento e do estado central do agente. O agregado do objeto resultante representa um objeto lógico, apesar de consistir em objetos do papel fisicamente diferentes. O padrão Objeto do Papel evita a explosão combinatória de classes, como

resultaria do uso de várias heranças para compor diferentes papéis em uma única classe [19]. No entanto, os clientes da classe UserAgentType provavelmente ficarão mais complexos, uma vez que trabalhar com um objeto por meio de uma de suas interfaces do papel implica uma leve codificação em comparação ao uso da interface fornecida pela classe UserAgentType. Por exemplo, os papéis de revisor e chair precisam ser explicitamente criados e ligados a objetos do agente do usuário, e o cliente tem de verificar se o objeto exerce o papel desejado antes da ativação explícita de algumas capacidades introduzidas por ele (Figura 59). Como conseqüência, o código da colaboração dos papéis é entrelaçado ao código não-colaborativo uma vez que os planos precisam implementar a conexão do papel. Além disso, os planos possuem referências explícitas aos papéis a fim de acessar os serviços específicos do papel. Isso aumenta o acoplamento do sistema.

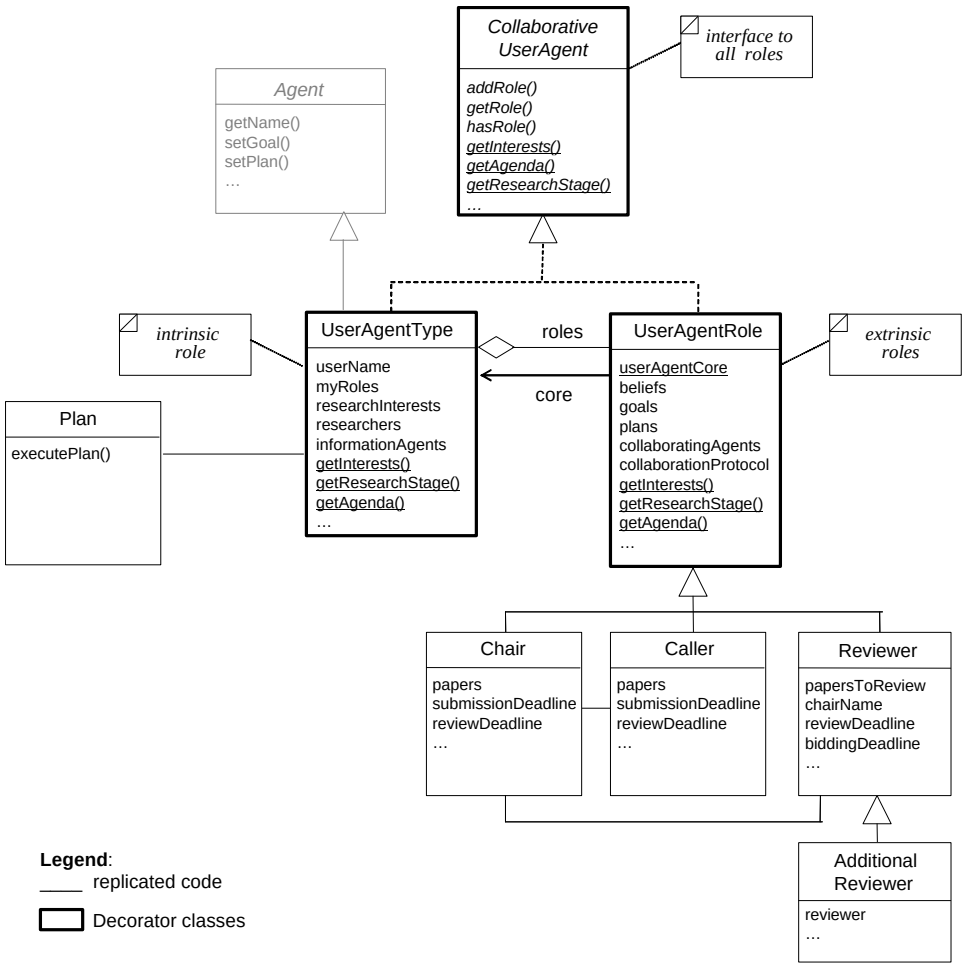


Figura 59. Papéis: o padrão Objeto do Papel com o padrão Decorador [146].

Kristensen e Osterbye [146] e outros autores [77, 121, 136] apontam as principais desvantagens a seguir:

- *Esquizofrenia do agente:* O comportamento do agente é distribuído por tipo de agente e seus papéis. O agente deve ser um objeto, mas, em vez disso, incorpora vários objetos, cada um com sua própria identidade (Seção 3.6.3). Isso viola o princípio de identidade (Seção 3.4.2), o que pode levar a muitos sintomas, incluindo delegação interrompida, assunções interrompidas e *dopplegangers* [121].
- *Mistura de interface* ou *delegações sucessivas*: A interface para todos os papéis deve ser fornecida em uma interface (CollaborativeUserAgent - Figura 59), aumentando a replicação do código e diminuindo a separação de concerns. Se isso não for feito, os objetos devem receber delegações sucessivas em tempo de execução para chamar comportamentos de papel específicos.
- *Composição do papel*: A versão *Decorador* do padrão Objeto do Papel não oferece suporte a referências diferentes, mas sobrepõe subconjuntos de decoradores. Ou seja, não oferece suporte ao princípio do papel de agregação (Seção 3.4.2).

Problema. Um tipo de agente pode exercer muitos papéis. Portanto, os papéis podem afetar vários tipos de agente [136]. Além disso, a ativação de um papel pode afetar as classes do plano. O código de ações não-colaborativas não deve ser misturado com o código de colaboração porque ele pode ser executado fora da colaboração. Conforme demonstrado no exemplo de motivação, os projetos orientados a objetos não oferecem suporte adequado a papéis do agente. Como separar os papéis das classes associados à funcionalidade básica do agente?

As forças são associadas aos princípios do papel apresentados na Seção 3.4.2, os princípios de abstratividade, agregação, dependência, dinamicidade, identidade, herança e multiplicidade. As forças a seguir são associadas ao problema:

- O projeto deve fornecer uma separação explícita entre o conhecimento intrínseco e o conhecimento extrínseco específico ao papel de um agente.
- Deve ser fácil desenvolver as ações colaborativas de um agente adicionando e removendo novos papéis de forma flexível.
- A solução deve minimizar a esquizofrenia do agente, melhorar a manutenção da interface e oferecer suporte à composição do papel.

Solução. Usar aspectos para melhorar a separação entre o tipo do agente e seus papéis. Um aspecto de papéis é usado para capturar o conhecimento extrínseco do papel. Os membros do conhecimento intrínseco básico fazem parte de uma classe que representa o tipo de agente, enquanto os membros do conhecimento extrínseco pertencem a um aspecto de papéis. Cada aspecto de papéis define a atividade do agente e o conhecimento dentro de uma determinada colaboração. Além disso, os relacionamentos e o contexto do papel residem na instância do aspecto para facilitar o suporte à *multiplicidade* do papel. Os aspectos Role podem afetar dinamicamente os diversos objetos (princípio da *dinamicidade*). Esses aspectos não existem por si só, sua criação sempre depende da instanciação de um tipo de agente (princípio da *dependência*). As especializações do papel têm suporte da herança do aspecto (princípio da *herança*).

Usando os aspectos Role, definimos quando e como o agente exerce o papel. Eles conectam os pontos de execução do programa (eventos) das classes do agente aos papéis correspondentes. Esses aspectos conseguem afetar alguns pontos de execução do agente – por exemplo chamadas a métodos das classes Agent – a fim de ativar um papel. Eles monitoram esses pontos de execução a fim de identificar quando o papel deve começar a ser exercido pelo agente. Em geral, quando o agente é criado, alguns papéis são acoplados a ele. Ademais, outros eventos específicos podem ser causados pela ativação dos papéis. As classes auxiliares são usadas para implementar os elementos do conhecimento extrínseco do papel, incluindo os objetivos, as crenças e os planos (Figura 60).

Estrutura. O padrão Papel possui três participantes principais e uma categoria cliente:

Principais Participantes:

- **Tipo de agente**

- implementa o papel intrínseco, ou seja, a funcionalidade básica, associada a um tipo de agente.

- **Aspecto Role**

- define um papel extrínseco do agente, incluindo suas ações e crenças simples, e as referências a crenças, objetivos e planos complexos.

- **Elemento do conhecimento extrínseco**

- representa um elemento do conhecimento extrínseco associado ao papel do agente – inclui um objetivo, uma crença e um plano.

Participantes cliente:

- **Elemento do conhecimento intrínseco**

- oferece o contexto a partir do qual um papel pode ser instanciado.

Na estrutura do padrão Papel (Figura 60), não há partes comuns para a instanciação do padrão. A realização do padrão requer:

1. A definição dos aspectos de papéis com ações e crenças simples.
2. A definição de classes da crença, classes do objetivo e classes do plano.
3. A especificação dos eventos associados ao tipo de agente e os elementos do conhecimento em que o aspecto Role precisa estar conectado à instância do agente.

O propósito dos aspectos Role é implementar os diferentes papéis do agente. Eles modularizam as atividades colaborativas específicas a um agente, separando-as da funcionalidade básica do agente. Todos os comportamentos específicos a papéis (extrínsecos) estão localizados no código-fonte do aspecto. Dessa forma, um aspecto

de papéis pode ser usado para separar de forma eficaz cada concern do papel. Esse aspecto contém os pointcuts que descrevem os eventos que devem motivar a ativação do papel. Contém os advices, associados a esses pointcuts, responsáveis pela ativação do papel. O aspecto Role possui duas partes principais: o aspecto em si e uma interface crosscutting. O aspecto possui as estruturas de dados que implementam o conhecimento extrínseco do papel. O comportamento do papel é colocado em conjunto com inter-type declarations que são adicionadas ao tipo de agente e advices que informam às classes do conhecimento intrínseco. Ou seja, um aspecto estático introduz a interface do comportamento específico a papéis à classe do tipo do agente e, em seguida, uma instância do aspecto adiciona ou informa a implementação desse comportamento a instâncias da classe do tipo do agente, dinamicamente e conforme necessário.

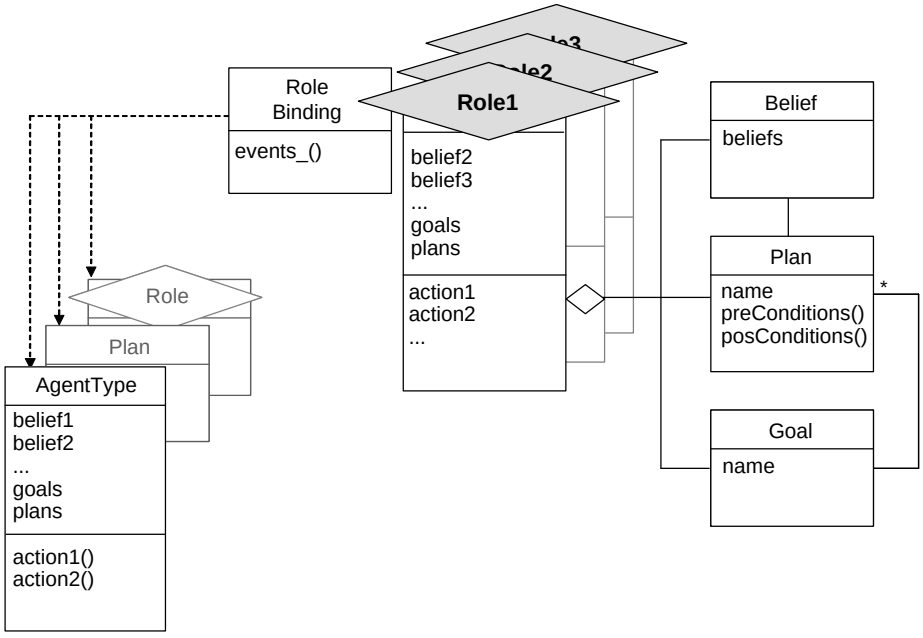


Figura 60. A visão estática do padrão Papel.

A interface crosscutting RoleBinding define quando o aspecto Role é acoplado às classes do agente. Ela define os join points nos elementos do conhecimento que devem disparar a ativação do papel. Contém um advice que executa depois de execuções de ações de classes do agente, ações de classes do plano

e outros aspectos associados ao agente (por exemplo, os aspectos de papéis). Essa abordagem a aspectos de papéis satisfaz quase todos os princípios do papel.

A Figura 61 ilustra a instanciação de padrões dos agentes de usuário no sistema Expert Committee. O aspecto Role afeta cerca de 4 classes e aspectos do agente nesse sistema. Todavia, na figura, alguns componentes foram omitidos, como o AdditionalReviewer; é uma outra subclasse de Reviewer. Ela mostra a classe UserAgent, a classe DistributionPlan, as outras seguem essencialmente o mesmo padrão. Como o papel de chair está relacionado ao papel de chamador, há um relacionamento entre os aspectos Chair e Caller. Eles são adicionados diretamente à classe UserAgent. No exemplo apresentado, eles estão ligados às instâncias UserAgent quando são criados. Como conseqüência, um pointcut é especificado para escolher as chamadas ao construtor da classe UserAgent. O aspecto Chair fornece ao agente ações, objetivos e planos que implementam as funcionalidades específicas a chair. Ele implementa as ações para colaborar com o papel de revisor. Contém, por exemplo, o plano e as ações associadas para distribuir propostas de revisão aos revisores e receber as respostas. De forma similar, o aspecto Reviewer introduz a capacidade de receber e julgar as propostas e enviar as respostas a chair.

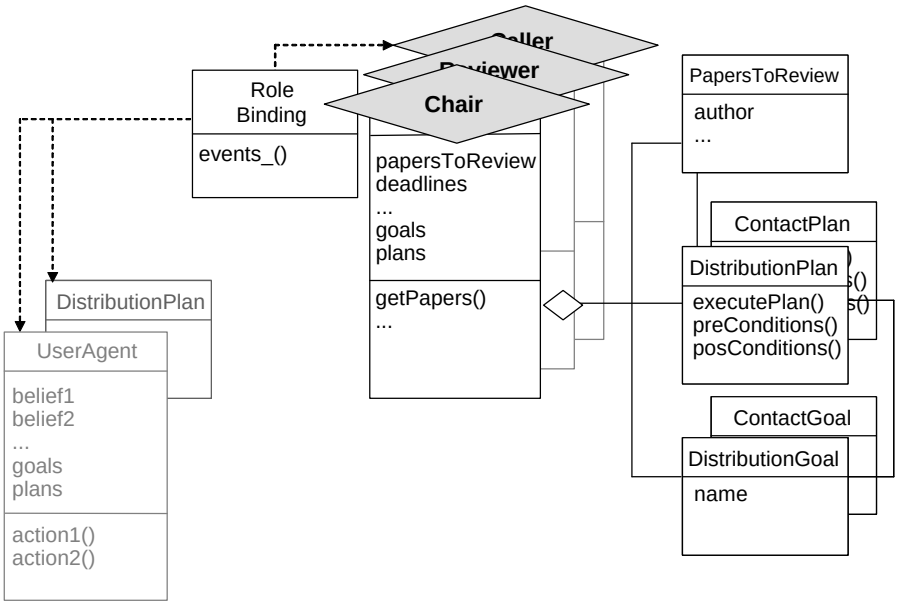


Figura 61. O padrão Papel para o agente do usuário do EC.

O aspecto Caller fornece a UserAgent a capacidade de enviar uma solicitação de busca para um agente de informação e de receber o resultado da busca. Ele é ativado no contexto do papel de chair (princípio de agregação). Um after advice startsCaller() está associado a recebimentos de métodos de busca (search(*)) e é responsável pelo envio da solicitação de busca quando o próprio agente não é capaz de encontrar as informações necessárias. Esse advice verifica os resultados dos métodos de busca de forma que o chamador seja ativado sempre que o resultado do método for nulo. Observe que os papéis são introduzidos de forma transparente e não-intrusiva. Observe que os diferentes subtipos de UserAgent podem herdar os aspectos de papéis acoplados à superclasse UserAgent (princípio de herança).

Dinâmica. Os cenários a seguir descrevem o comportamento dinâmico do padrão Papel.

Cenário I – Ligando uma instância do papel a uma instância do agente, ilustrado pela Figura 62, apresenta o comportamento do padrão quando o aspecto Role é instanciado na criação de um novo agente:

- O agente é criado pela chamada do construtor da classe Agent.
- O aspecto Role detecta a criação do agente interceptando a chamada ao construtor do agente.
- O advice no aspecto de papéis inicializa o papel e seus atributos internos.

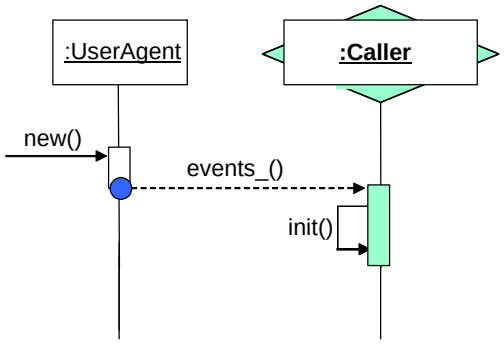


Figura 62. Padrão Papel: ligando uma instância do papel a uma instância do agente.

Cenário II – Ativando a instância do papel nas diferentes situações. O aspecto Role pode ser ligado a um agente em situações muito específicas. Por exemplo, o aspecto Caller é ativado quando o papel de chair não contém um determinado perfil de revisor. Esse aspecto implementa a colaboração com um agente Information que é responsável pela busca das informações necessárias em um banco de dados locais. A Figura 63 ilustra esse cenário:

- O papel de chair tenta encontrar um determinado perfil de revisor.
- O aspecto Caller detecta a chamada do método searchProfile().
- O advice no aspecto Caller não retorna nenhum perfil para esse revisor.
- O papel de Caller chama suas ações colaborativas (implementadas por seus métodos internos) a fim de colaborar com um agente de informação.
- O aspecto Interaction do papel de Caller envia uma mensagem de solicitação ao agente de informação.

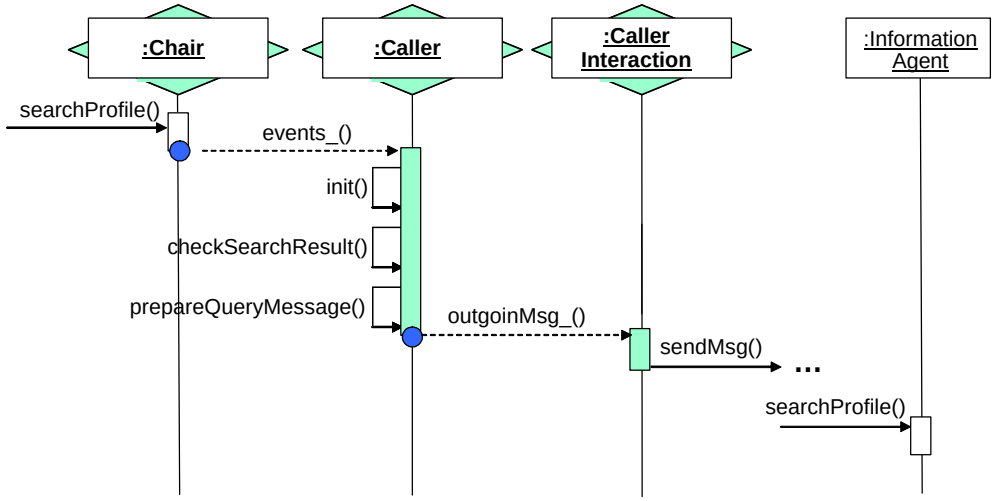


Figura 63. Padrão Papel: ativando um papel quando a informação não é encontrada.

Conseqüências. O padrão Papel satisfaz quase todos os princípios do papel. Ele oferece suporte a instâncias do agente que exercem mais de um papel ao mesmo tempo. Esses papéis podem ser independentes (multiplicidade do papel) ou podem ser agregados (composição do papel). A multiplicidade do papel pode ser implementada

indexando os papéis por contexto em que aparecem. Também tem os benefícios a seguir em relação ao padrão Objeto do Papel:

- *Ausência de mistura da interface*. O padrão não requer que as interfaces de todos os papéis sejam fornecidas em uma única interface como no padrão Papel do Objeto. A interface intrínseca do agente não é misturada com cada interface do papel potencial. O comportamento extrínseco é acessível sem delegações sucessivas.
- *Minimização da esquizofrenia do agente*. A maior parte dos comportamentos específicos a agente reside no objeto; somente o relacionamento do papel e o contexto do papel residem no aspecto. Não há classes intermediárias como no padrão Papel do Objeto.
- *Melhor separação de concerns*. Os tipos de agente não têm referência a seus papéis. O grau de acoplamento do SMA também aumenta.
- *Menor número de componentes.* O padrão Papel não requer classes adicionais como AgentRole e AgentInterface, que são participantes centrais no padrão Papel do Objeto.
- *Falta de replicação do código.* Não há necessidade de replicação de todas as assinaturas de métodos do papel em uma AgentInterface. Como conseqüência, facilita as atividades de manutenção.
- *Plug and Play*. Com a evolução do sistema multiagentes, novos papéis do agente precisam ser adicionados de forma transparente a um tipo de agente. Os desenvolvedores de agentes somente precisam adicionar novos pointcuts para associar o tipo de agente ao novo aspecto de papéis.

Por outro lado, o uso do padrão Papel possui algumas desvantagens:

- *Refatoração necessária*. Em algumas circunstâncias, a realização do padrão Papel afeta a estruturação das classes Kernel. Isso requer a reestruturação dessas classes a fim de expor join points adequados. Por exemplo, precisamos garantir que cada método que busca o perfil do usuário retorne um valor de objeto de forma que o aspecto possa capturar a resposta do usuário e verificar

se o valor retornado foi nulo. Além disso, extraímos o código dos métodos de planos existentes para criar um novo método para expor um join point no nível do método. As ferramentas para ajudar o processo de reestruturação facilitariam a introdução de aspectos em um sistema existente.

- *A descrição de aspectos de papéis depende de classes principais específicas*. Os nomes das classes do conhecimento aparecem na definição dos aspectos de papéis. Portanto, a descrição de um aspecto de papéis não pode ser diretamente aplicada a outras classes principais. A abordagem RoleEP [240] resolve os pontos fracos desse padrão.

Variantes. *Papéis intrusivos*. Essa solução requer a instância do aspecto para adicionar o comportamento do papel, informando aos membros do papel que já existem em uma instância central [136]. Ela requer que o comportamento do papel esteja disponível na interface da classe do agente, uma vez que só os membros existentes podem ser modificados. O variante diminui a separação dos concerns entre o agente e seus papéis.

Papéis estáticos. Esse variante introduz o comportamento do papel em uma classe do agente, em vez de no nível de instância do agente [136]. Isso significaria que todas as instâncias de um dado tipo de agente exercem os mesmos papéis, violando a dinamicidade do papel.

Aspectos de Papéis como conectores. Os papéis e os tipos de agente são classes nessa solução. Um aspecto estático integra ou compõe as hierarquias das classes do papel e as hierarquias dos tipos de agente. Essa abordagem parece ser extensível [136]. Contudo, ela requer três níveis de componentes (objetos centrais – representando os agentes, os papéis e aspectos) e fica complexa quando há muitas dependências entre papéis e objetos centrais. Além disso, ela não satisfaz o princípio da identidade.

Usos conhecidos. Kendall [136] usa e implementa os três variantes acima usando AspectJ. Em seu modelo de papel, um objeto possui métodos/atributos intrínsecos centrais e um papel que adiciona métodos/atributos extrínsecos e fornece perspectivas que podem ser usadas por outros objetos. Kendall recomendou uma abordagem alternativa: (i) introduzir a interface para o comportamento específico a papéis na

classe central; (ii) informar a implementação do comportamento específico a papéis a instâncias da classe central; (iii) adicionar relacionamentos e contextos do papel na instância do aspecto. Entretanto, a solução de Kendall requer que os clientes conheçam os aspectos de papéis; aumentando, então, o acoplamento do sistema. Ubayashi e Tamai [240] propuseram a abordagem RoleEP (Role Based Evolutionary Programming) que é uma realização do padrão Papel. Epsilon/J é um framework que oferece suporte à abordagem RoleEP. Um objeto RoleEP torna-se um agente, ligando-se a um papel definido no ambiente e adquire as funções de colaboração dinamicamente. No entanto, sua proposta tem algumas limitações: o código de ligação do papel é entrelaçado ao código de agente não-colaborativo, e a classe do tipo de agente precisa implementar alguns métodos de uma interface RoleEP. Finalmente, implementamos o padrão Papel no sistema Portalware (Capítulo 4) e no sistema EC.

Padrões relacionados. O padrão Papel é uma alternativa melhor do que a combinação do padrão *Objeto do Papel* [19] com o padrão Decorador [79], que é a melhor solução conhecida orientada a objetos para a implementação de papéis do agente [77, 136, 146]. O padrão Papel usa o padrão Kernel do Agente uma vez que os papéis modularizam o conhecimento extrínseco e afetam as classes do kernel. Esse padrão também está relacionado a todos os outros padrões em nossa linguagem de padrões, uma vez que pode refinar aspectos de agência e aspectos adicionais a um contexto do papel. Um modelo do papel é fornecido no padrão *Burocracia* [203, 204]. Esse padrão é normalmente encontrado em sistemas de software [204], mas também captura a estrutura das burocracias humanas [136].

O padrão Papel pode ser usado juntamente com alguns padrões existentes em estratégias de coordenação e colaboração específicas. Alguns exemplos clássicos são: Reunião, Bloqueio, Mensageiro, Facilitador e Grupo Organizado [10].O padrão *Blackboard Reflexivo*, apresentado em outro trabalho [221], oferece suporte à colaboração guiada por eventos. Hayden et al. propõem um sistema de padrões arquiteturais para a coordenação multiagentes [117]. Lea [154] propõe um conjunto de padrões de concorrência que podem ser usados para coordenar vários papéis. De fato, usamos o padrão *Objeto Compartilhado* no sistema Portalware (Capítulo 4) para

coordenar as atividades do papel. Finalmente, dependendo da complexidade do papel, a implementação de AspectJ do padrão Papel requer o uso de idiomas avançados, como o *Método Pointcut* e *Pointcut Abstrato* [114].

5.8
O padrão Mobilidade

Intenção. O padrão Mobilidade desacopla o comportamento da mobilidade da funcionalidade básica do agente e outros concerns do agente.

Também conhecido como. *Padrão Deslocamento, Padrão Migração.*

Contexto. Vários tipos de agentes e papéis podem ter a propriedade de mobilidade. Durante a execução de seus planos, um agente móvel pode se mover de um ambiente em uma rede para outro a fim de alcançar seus objetivos. Muitas facetas de uma estratégia de mobilidade devem ser consideradas [240], incluindo a especificação: dos elementos do agente (papéis e tipos de agente) que são móveis, das circunstâncias em que o agente precisa se mover, da partida para ambientes móveis, do retorno ao ambiente *host* e do itinerário do agente. Essas facetas da estratégia de mobilidade devem sem diretas em relação às funcionalidades básicas dos agentes.

Exemplo de motivação. Os agentes do usuário do EC precisam se mover em determinadas circunstâncias. Por exemplo, quando um agente está exercendo o papel de chair, ele precisa consultar os perfis dos revisores a fim de otimizar a distribuição de trabalhos em termos dos interesses de cada um. Se o perfil de um revisor não estiver disponível, ele colabora com um agente de informação e solicita a esse agente as informações sobre um determinado revisor. O agente de informação controla o banco de dados local e consegue buscar o perfil. Entretanto, se as informações não estiverem disponíveis no banco de dados, o agente da chair precisa se mover e tentar encontrar o perfil que está faltando nos ambientes remotos. O agente atribui a tarefa de busca aos agentes de informação dispersos na Internet ao fazer a migração por hosts.

Há diversos frameworks orientados a objetos que oferecem suporte à implementação do concern da mobilidade em agentes de software, como Aglets [120,

152] e JADE [20]. Ubayashi e Tamai [240] apresentam abordagens orientadas a objetos para a separação do concern de mobilidade da funcionalidade do agente e de outros concerns, como a colaboração. Eles apresentam o sistema de padrões do projeto de Aridor e Lange [10, 152] como uma das melhores soluções orientadas a objetos. Esses padrões de projeto foram implementados no framework Aglets. Nessa abordagem, o padrão Itinerário possui as informações para a migração – ele encapsula em uma instância da classe Itinerary. Para tornar um tipo de agente móvel, a classe do tipo de agente é definida como uma subclasse da classe Agent.

Todavia, Ubayashi e Tamai [240] defendem que essa solução baseada em padrão possui algumas limitações para fornecer o isolamento do concern de mobilidade. Apesar de as funções de mobilidade (código para a migração em hosts) serem separadas das funções de colaboração (ações do papel), elas são misturadas com as ações básicas do agente dentro das classes do kernel do agente. Se um agente precisar ter vários papéis ou ações de mobilidade, seu código será mais complexo [240]. Como todos os agentes móveis precisam estender a classe Aglets, a classe do tipo de agente possui uma referência explícita à classe Aglet. Também possui uma referência explícita a uma instância da classe Itinerary. As ações básicas do agente estão entrelaçadas com chamadas explícitas ao método roam().

Esses problemas não são específicos aos padrões Aridor e Lange. Os frameworks de mobilidade normalmente impõem nas classes do agente a extensão de classes específicas à mobilidade, a extensão de métodos abstratos dessas classes de mobilidade e a chamada de métodos de mobilidade nas ações básicas do agente. A Figura 64 ilustra problemas similares ao usar o framework JADE em agentes dos usuários do EC. Observe que o código de mobilidade afeta os diversos métodos e as classes de papéis e planos. Além disso, outros revisores têm de herdar o comportamento de mobilidade mesmo quando não será realizada nenhuma ação de mobilidade no contexto desse papel. Supondo que algumas instâncias de outros revisores serão móveis e outras serão estáticas, um nível de herança precisa ser adicionado ao sistema a fim de separar o outro revisor móvel e estático nas diferentes classes. Ademais, todas as classes do agente precisam ser serializadas nos ambientes Java, ou seja, as classes do agente precisam implementar a interface Serializável para

fins de mobilidade. A abordagem baseada em *mixins* [26] é normalmente usada para tratar esse tipo de problema. Nesse caso, as funções solicitadas para um gerenciador podem ser descritas em uma superclasse. Se um agente possui muitos papéis, o agente precisa herdar estaticamente as superclasses correspondentes. Portanto, o código do programa precisa ser modificado sempre que os papéis solicitados para um agente são adicionados ou excluídos. Além do mais, várias heranças não são permitidas em muitas linguagens de programação, como Java.

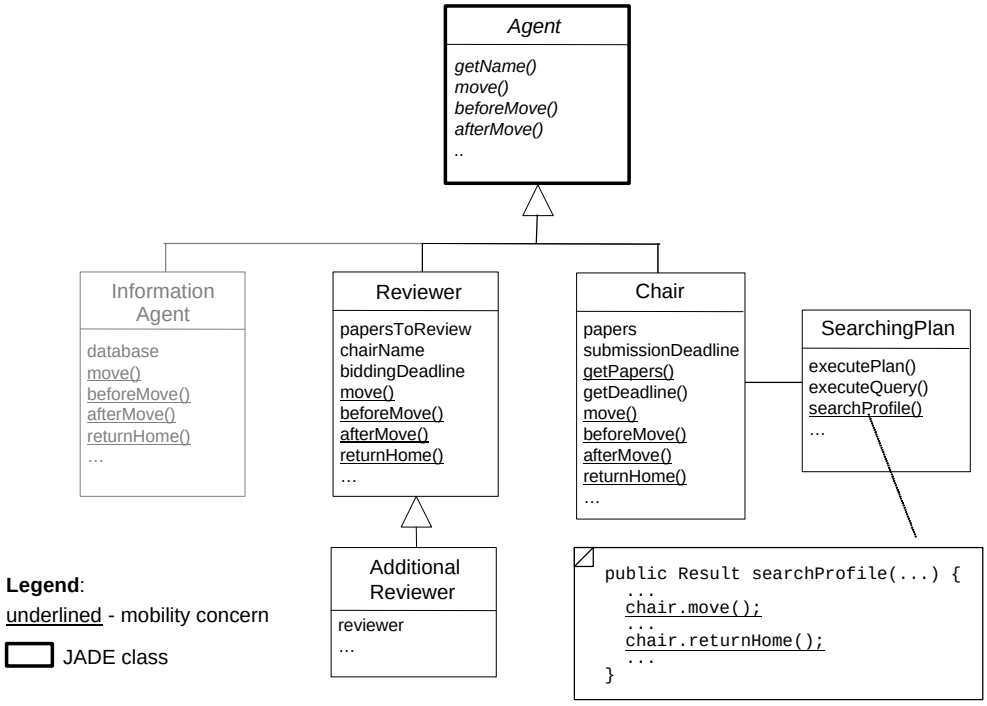


Figura 64. Concern de mobilidade: afetando papéis, planos e tipos de agente.

Problema. O fato de agentes e seus papéis mudarem sua localização não deve afetar sua funcionalidade básica. O uso de soluções orientadas a objetos tende a entrelaçar o código da mobilidade dentro da funcionalidade básica do agente. Como conseqüência, é difícil entender colaborações entre agentes e deslocamentos de agentes individuais de uma forma geral, o que, por sua vez, diminui a manutenibilidade e reusabilidade do sistema. É difícil definir funções e comportamentos de agentes de forma elegante porque não há suporte para defini-los independente do código de mobilidade. Além disso, é complicado estender o código relacionado a cada concern. Como os agentes incorporam a propriedade de

mobilidade sem ter sua funcionalidade básica entrelaçada com o concern de mobilidade? As forças a seguir são associadas a esse problema:

- O projeto deve oferecer suporte à introdução uniforme dos comportamentos de mobilidade dos tipos de agente e papéis.
- Quando um papel se move, seu agente associado precisa ser movido, uma vez que o papel depende das classes do kernel de agente (Seção 5.3).

Solução. Usar aspectos para melhorar a separação entre o concern de mobilidade e outros concerns do agente. O aspecto de mobilidade é usado para modularizar a estratégia de mobilidade que afeta muitas partes do agente do software. Um aspecto de mobilidade separa o comportamento de mobilidade de elementos de conhecimento do agente, como ações e planos, tipos de agente e aspectos de papéis. Ele implementa não apenas as ações de mobilidade básicas, mas também a especificação de quais tipos de agente ou papéis são móveis, a declaração das circunstâncias de deslocamento, as chamadas de métodos de partida e retorno e o controle do seu itinerário.

Os aspectos de mobilidade definem como e quando os agentes se deslocarão para outros ambientes. Esses aspectos conseguem afetar alguns pontos de execução do agente – por exemplo chamadas a métodos nas classes Plan – e alteram sua execução normal a fim de verificar a necessidade de transferir o agente para outro host. O próprio aspecto chama os métodos responsáveis pela implementação das ações de mobilidade. Nenhum código de mobilidade permanece nos outros aspectos e classes do agente.

Estrutura. O padrão Mobilidade possui dois participantes principais e duas categorias cliente:

Principais Participantes:

- **Elemento móvel**

- representa o elemento móvel – pode ser um papel ou um tipo de agente.

- **Aspecto Mobility**

- implementa todo o comportamento de mobilidade.

Participantes cliente:

- **Elemento do conhecimento**

- fornece o contexto a partir do qual as ações de mobilidade são disparadas – esse elemento não possui código de mobilidade.

- **Framework de mobilidade**

- é parte do framework de mobilidade usado e fornece os serviços de mobilidade.

A estrutura do padrão Mobilidade (Figura 65) possui algumas partes que são comuns a todas as instanciações potenciais do padrão e outras partes que são específicas a cada instanciação. As partes comuns são:

1. Os eventos genéricos que disparam a partida do agente para ambientes remotos.
2. O protocolo de mobilidade geral:
    a. os eventos são escolhidos
    b. as condições são verificadas e
    c. as ações de mobilidade são chamadas.
3. As ações de mobilidade genéricas:
4. O objeto itinerário.

As partes específicas para cada instanciação do padrão são:

5. Os eventos específicos associados a um contexto específico (tipo de agente ou papel) quando o agente precisa ser movido – as classes ou aspectos precisam ser observados.
6. As ações de mobilidade específicas:

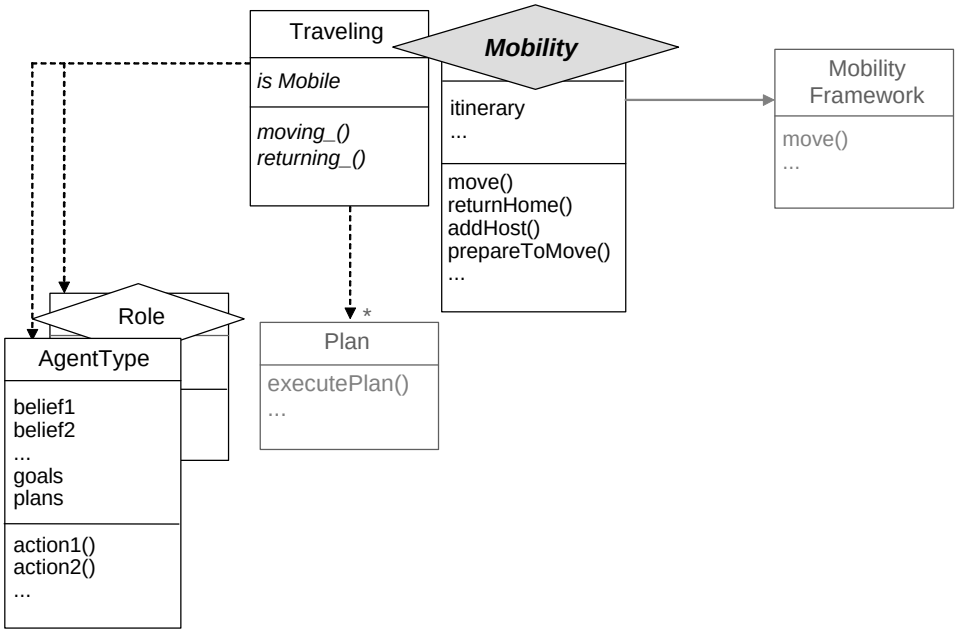


Figura 65. A visão estática do padrão Mobilidade.

O propósito dos aspectos Mobility é desacoplar os comportamentos de mobilidade dos diferentes papéis do agente. Eles modularizam a estratégia de mobilidade de um agente ou de um papel, separando-a de sua funcionalidade básica. Todas as estratégias de mobilidade estão localizadas no código-fonte do aspecto. Esse aspecto contém os pointcuts que descrevem os eventos que devem levar o agente a se deslocar para um ambiente remoto. Os aspectos Mobility também contêm os advices associados a esses pointcuts. Os advices são responsáveis por verificar a necessidade de migração do agente e pela chamada das ações de mobilidade. O aspecto Mobility possui duas partes principais: o aspecto em si e uma interface crosscutting. Ele possui as estruturas de dados e os métodos que controlam o itinerário do agente. O aspecto também implementa os métodos para deslocamento até um novo ambiente e retorno ao *host* original.

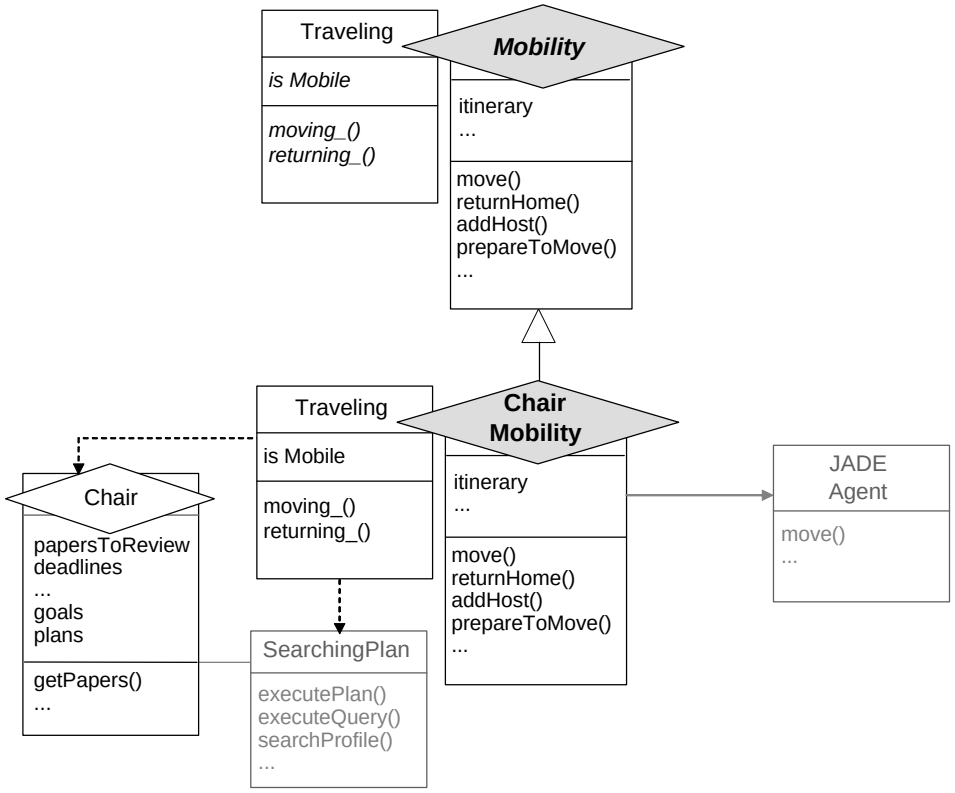


Figura 66. O padrão Mobilidade do papel de Chair.

A interface crosscutting Deslocamento define quais tipos de agente e papéis são móveis. Também define quando esses elementos do SMA devem se mover. Define os pointcuts nas classes do agente ou aspectos de papéis que devem disparar a migração do agente. Há dois pointcuts: o primeiro especifica quando o agente deve se mover para outro ambiente, o segundo especifica quando o agente deve voltar para o ambiente original. Ele contém dois advices que são executados depois da execução das ações em subclasses Agent, ações em subclasses Plan ou ações em aspectos de papéis. Dependendo do framework de mobilidade usado em um determinado SMA, a interface também descreve as classes que são serializadas ou devem implementar interfaces de framework específicas (representadas por “is Mobile” na Figura 65). Como conseqüência, as classes do agente não estão entrelaçadas com o código de mobilidade, melhorando, assim, sua manutenibilidade e reusabilidade. Os pointcuts são declarados como abstratos porque precisam ser redefinidos para papéis e tipos de agente específicos.

O aspecto Mobility no sistema EC afeta cerca de 7 classes e aspectos do agente. A Figura 66 ilustra a instanciação de padrões do papel de chair nesse sistema. Ela mostra o aspecto ChairMobility afetando o aspecto Chair e a classe SearchingPlan. O aspecto Mobility afeta esse plano porque quando o método searchProfile() retorna nulo, o agente precisa se mover. O pointcut que se move define as execuções do método searchProfile() como os join points de interesse do aspecto ChairMobility. Também afeta o aspecto Chair porque o agente precisa fazer uma migração dependendo dos resultados de algumas ações de papel internas, como o método getPapers().

Dinâmica. Alguns cenários são descritos a seguir a fim de ilustrar o comportamento dinâmico do padrão Mobilidade. Os cenários mostram como os aspectos de mobilidade movem os agentes e os papéis de forma transparente.

Cenário I – Movendo um papel do agente sempre que seu plano não consegue encontrar uma informação, ilustrado pela Figura 67, apresenta o comportamento do padrão quando o aspecto de mobilidade detecta a necessidade de mover o agente para um ambiente remoto:

- O plano de busca da chair é chamado.
- O aspecto ChairMobility detecta a execução do método searchProfile() na classe SearchingPlan.
- Esse aspecto reúne as informações necessárias do contexto do plano, ou seja, o valor de retorno do método searchProfile().
- O aspecto de mobilidade detecta a necessidade de mover o agente que exerce o papel de chair.
- O aspecto chama o método prepareToMove() para executar as últimas ações antes da migração do agente.
- O aspecto ChairInteraction intercepta a execução desse método para notificar os demais agentes no ambiente para o qual o agente está se movendo. [5]

[5] Essa funcionalidade é às vezes fornecida pelo framework de mobilidade.

- O aspecto ChairMobility chama o método para mover fisicamente o agente para o ambiente remoto.
- O método do framework de mobilidade (JADEAgent) implementa o movimento do agente.

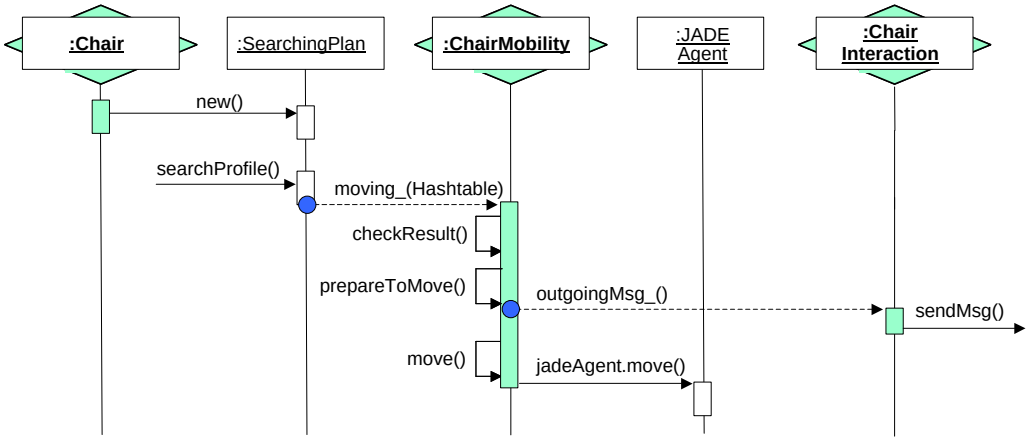


Figura 67. Padrão Mobilidade: movendo quando um plano não consegue encontrar informações.

Cenário II – Movendo um agente do papel sempre que sua ação interna não consegue encontrar uma informação, ilustrado pela Figura 68, apresenta o comportamento do aspecto Mobility quando ele detecta a necessidade de mover o papel do agente para um novo ambiente:

- O método getPapers() é chamado.
- O aspecto ChairMobility detecta a execução do método getPapers() no aspecto Chair.
- Esse aspecto reúne as informações necessárias do contexto do plano, ou seja, o valor de retorno do método getPapers().
- O aspecto ChairMobility detecta a necessidade de mover o agente que exerce o papel de chair.
- O aspecto chama o método prepareToMove() para executar as últimas ações antes da migração do agente.
- O aspecto ChairInteraction intercepta a execução desse método para notificar os demais agentes no ambiente para o qual o agente está se movendo.

- O aspecto chama o método para mover fisicamente o agente para o ambiente remoto.

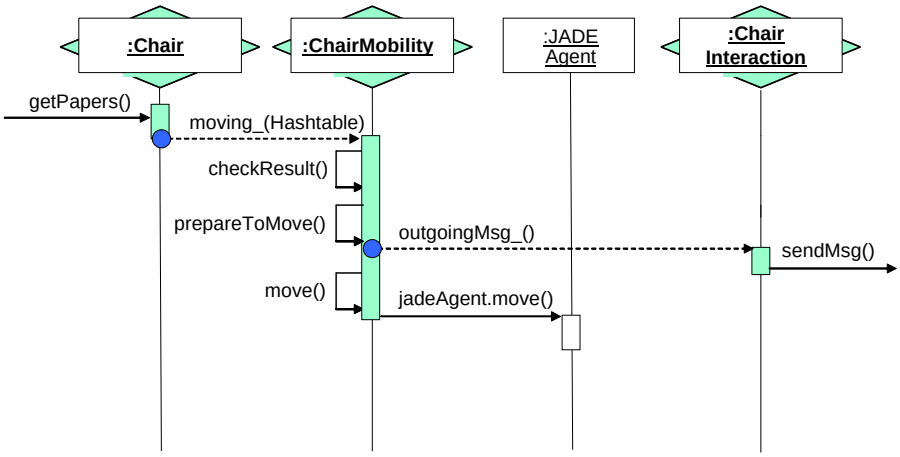


Figura 68. Padrão Mobilidade: movendo quando uma ação do papel não consegue encontrar informações.

A dinâmica do padrão é semelhante ao caso em que uma ação de um tipo de agente dispara o deslocamento do agente.

Conseqüências. A solução de padrão fornece os benefícios a seguir:

- *Melhor separação de concerns*. O concern de mobilidade é modularizado em aspectos. As classes do agente básicas não implementam o comportamento de mobilidade e não são alteradas a fim de implementar a estratégia de mobilidade. O código dos aspectos de papéis também se mistura ao código de mobilidade.
- *Flexibilidade.* As estratégias da mobilidade exploradas na aplicação do SMA podem variar à medida que o sistema se desenvolve. O projeto é mais flexível a fim de oferecer suporte à alteração da estratégia de mobilidade usada.
- *Transparência.* O concern de mobilidade pode ser adicionado ou removido de forma transparente aos/dos agentes estáticos uma vez que seu código não precisa ser alterado.
- *Melhor suporte para manutenção*. Há duas outras questões no protocolo de mobilidade que normalmente mudam: (i) os pontos de execução do agente que

disparam a migração do agente e (ii) os eventos que disparam o retorno do agente para o local original. Como a definição desses pontos e eventos está localizada em um único módulo, o aspecto Mobilidade, a manutenibilidade do agente é melhorada.

Apesar de o concern de mobilidade ser completamente definido separado de outros concerns do agente, o uso do padrão impõe alguns problemas ao projetista do SMA:

- *Refatoração necessária*. Em algumas circunstâncias, a realização do padrão Mobilidade requer a reestruturação do código-base associado aos outros componentes do agente a fim de expor join points adequados. Ocorre uma questão semelhante com o padrão Autonomia e o padrão Papel.
- *A descrição dos aspectos Mobility depende das classes Core específicas*. Os nomes das classes do agente e dos aspectos de papéis aparecem na definição dos aspectos Mobility. Portanto, a descrição de um aspecto de mobilidade não pode ser diretamente aplicada a outras classes Core. Esse problema é similar àquele encontrado no padrão Papel (Seção 5.7).

Variantes. *Mobilidade reflexiva*. Esse variante é similar à solução orientada a aspectos apresentada aqui. A diferença está no uso dos metaobjetos como alternativa aos aspectos. No entanto, essa abordagem requer um protocolo metaobjeto [165] que normalmente introduz alterações na máquina virtual. Além disso, as soluções reflexivas não oferecem suporte à composição dos metaobjetos de mobilidade com outros concerns. Como a complexidade do agente aumenta, bons mecanismos de composição são essenciais para a manutenibilidade e reusabilidade do sistema.

Usos conhecidos. Implementamos o padrão Mobilidade no sistema Portalware (Seção 4.1) e no sistema EC. O framework Epsilon/J, que oferece suporte à abordagem RoleEP [240], implementou o variante reflexivo. A abordagem RoleEP só oferece suporte à mobilidade do agente e à mobilidade do papel, e os programadores de agente têm de estender várias interfaces Epsilon/J e classes abstratas. Em RoleEP, o uso de *operação de ligação* [240] elimina a necessidade de processos de

combinação no estilo POA. As inter-type declarations em AspectJ podem ser substituídas adicionando métodos de papel pela operação de ligação. Contudo, processos de combinação de advices não correspondem a qualquer construto de modelo em RoleEP. Esse é um ponto fraco de RoleEP e reduz a capacidade de evitar a duplicação de código. A partir do ponto de vista da evolução estática, o processo de combinação de advices é muito útil porque evita a duplicação de código.

Padrões relacionados. O padrão Mobilidade está relacionado ao padrão Kernel do Agente (seção 5.3) uma vez que o aspecto de mobilidade escolhe join points em classes do plano e classes do tipo de agente. Além disso, esse aspecto afeta várias classes do conhecimento a fim de serializá-las. Nesse sentido, o padrão Mobilidade afeta quase todas as demais classes do padrão, uma vez que todas as classes do agente precisam ser serializadas para fins de mobilidade. Conforme mostrado anteriormente, o padrão Mobilidade está relacionado a: (i) o padrão Papel quando os papéis são móveis e (ii) o padrão Interação para notificar os agentes locais de uma migração de agente.

O padrão Mobilidade pode estar combinado ao padrão Itinerário[10] a fim de registrar as ações e planos executados em cada *host* visitado. Também pode ser usado em conjunto com estratégias de mobilidade específicas, como Forwarding e Ticket [10]. O padrão Mobilidade é potencialmente usado em conjunto com alguns padrões existentes para a coordenação e a colaboração remota específicas. Alguns exemplos clássicos são: Reunião, Bloqueio, Mensageiro, Facilitador e Grupo Organizado [10]. O padrão *Blackboard reflexivo*, apresentado em outro trabalho [221], oferece suporte à mobilidade com base em espaços de tuplas distribuídos. Yoshioka et al. [250] propõem um sistema de padrões para implementar políticas de segurança para agentes móveis que podem ser combinados com o padrão Mobilidade.O padrão Mobilidade pode ser conectado a vários padrões clássicos, como: (i) o padrão Factory Method [79] – permite que os manipuladores de migração real e virtual sejam instanciados conforme necessário em tempo de execução, (ii) o padrão Objeto Ativo – trata do acesso a recursos compartilhados [153], (iii) o padrão Visitante – usado para a configuração remota [79] e (iv) o padrão Proxy – usado em questões de migração

virtual [79]. Finalmente, a implementação de AspectJ do padrão Mobilidade requer o uso de idiomas avançados, como o *Método Pointcut* e *Pointcut Abstrato* [114].

5.9
O padrão Aprendizagem

Intenção. O padrão Aprendizagem modulariza o protocolo de aprendizagem. Ele separa não só os algoritmos de aprendizagem, mas também o processo de coleta de informações a fim de oferecer suporte ao processo de aprendizagem, desacoplando a estrutura básica do agente do protocolo de aprendizagem. Especifica como extrair informações dos diferentes componentes do agente que são necessários para permitir a aprendizagem do agente. Ele conecta as classes do agente aos componentes de aprendizagem específicos, tornando a funcionalidade básica do agente transparente em relação às particularidades dos algoritmos de aprendizagem em uso.

Também conhecido como. *Padrão Protocolo de Aprendizagem.*

Contexto. Os agentes cognitivos aprendem com base em sua experiência como resultado de suas ações, seus erros, as interações sucessivas com o mundo externo e a colaboração com outros agentes [39, 172, 207]. A separação do protocolo de aprendizagem é necessária para facilitar a manutenção e reutilização dos componentes do agente.

Exemplo de motivação. A fim de aprender as preferências do usuário, os agentes de usuário no sistema EC supervisionam suas ações, a interação com os componentes dos diversos ambientes e suas colaborações interagente. Por exemplo, eles observam as mensagens trocadas entre revisores e chairs. Esses agentes usam duas técnicas de aprendizagem diferentes: Temporal Difference Learning (TD-Learning) [172] e Least Mean Squares (LMS) [172]. O primeiro é necessário para o contexto do papel de revisor e o último no contexto do papel de chair. Os revisores aplicam TD-Learning para aprender as preferências do usuário nos assuntos de que gosta ou não de revisar de acordo com as interações com o usuário. A chair implementa LMS para aprender sobre as preferências do revisor com base nas interações com os agentes revisores.

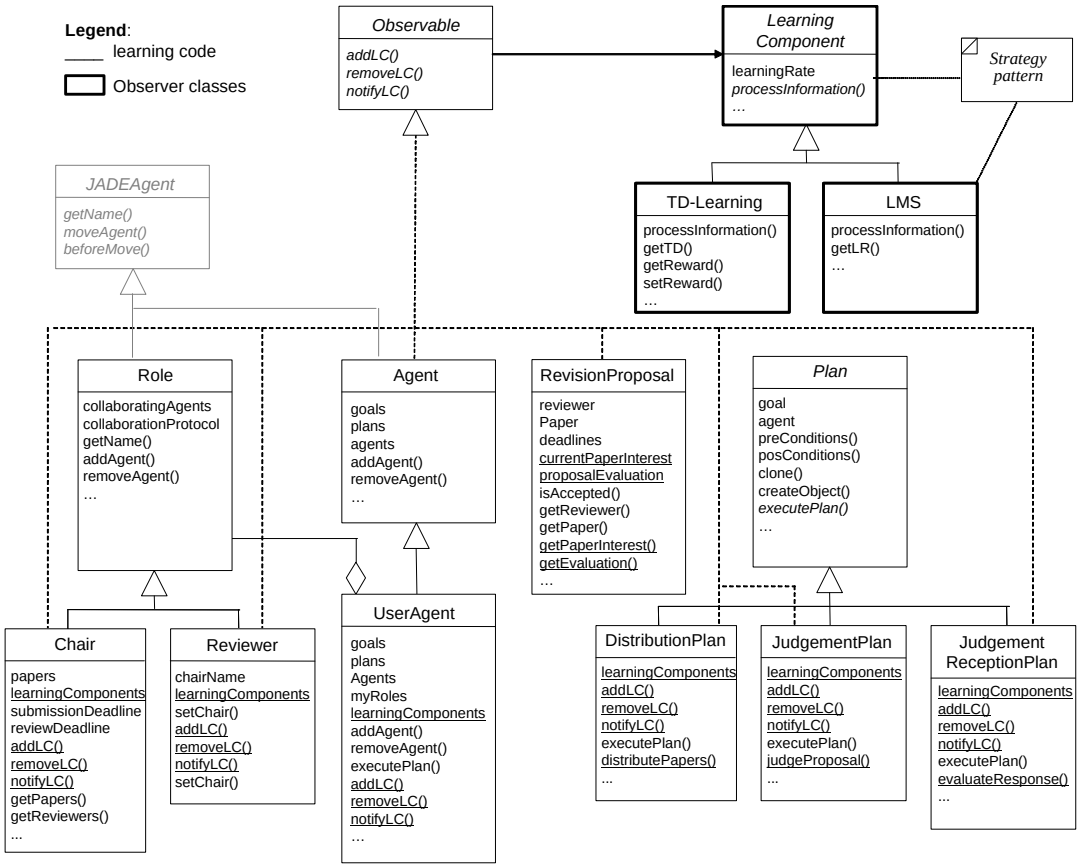


Figura 70. Aprendizagem: o padrão Observador com o padrão Estratégia.

Uma abordagem orientada a objetos flexível para projetar o protocolo de aprendizagem baseia-se na combinação do padrão Observador com o padrão Estratégia [137, 211, 212]. O padrão Observador [79] é útil para implementar o mecanismo para o monitoramento de eventos, enquanto o padrão Estratégia [79] é necessário para torná-lo flexível em relação às estratégias de aprendizagem. Considere um exemplo concreto dessa abordagem no contexto do sistema EC, conforme demonstrado na Figura 70. Nesse sistema, o padrão Observador é usado para notificar os componentes de aprendizagem sobre os eventos relevantes que disparam o processo de aprendizagem. As operações nas classes Plan, Agent, Role e Belief são monitoradas para fornecer ao componente de aprendizagem as informações contextuais e dar início ao processo de aprendizagem. O componente Observável é uma interface e uma classe não-abstrata porque as classes observáveis já estendem uma classe abstrata (JADEAgent). Muitas linguagens de programação orientadas a

objetos não oferecem suporte a várias heranças. Conforme mencionado anteriormente, Java, a linguagem orientada a objetos mais comum e expressiva para a implementação de SMAs [22, 20, 181], não oferece suporte a esse recurso. As classes Agent e Role não implementam a interface Observável porque a aprendizagem não é uma propriedade de agência (Seção 3.3). Alguns papéis e tipos de agente são monitorados pelos mecanismos de aprendizagem.

A classe LearningComponent implementa o padrão Estratégia e representa uma família de algoritmos diferentes que implementam as técnicas de aprendizagem. As subclasses de LearningComponent podem de alguma forma variar independente das classes do agente que as usam. As subclasses TD-Learning e LMS implementam os algoritmos de aprendizagem específicos. Todavia, a introdução do protocolo de aprendizagem possui um grande impacto na estrutura do agente. Ele afeta as diferentes classes do agente e aspectos. Conforme mostrado na figura, o código para a implementação desse padrão se espalha pelas hierarquias de classes de um agente de software. Todos os participantes (ou seja, Chair, Revisor, UserAgent, RevisionProposal, os planos) devem implementar o mecanismo de observação e, conseqüentemente, possuir o código de aprendizagem. Adicionar ou remover o código de aprendizagem das classes requer alterações nessas classes. A alteração no mecanismo de notificação (como alternar entre os modelos de push e pull [79]) requer alterações em todas as classes participantes.

Problema. Os padrões de projeto orientados a objetos normalmente têm vários efeitos indesejáveis ao tratar da propriedade de aprendizagem. O concern de aprendizagem afeta naturalmente as hierarquias de classes, perdendo sua modularidade. Ele afeta os tipos de agente, os planos, papéis, crenças e outros objetos. Essas classes precisam ser interligadas ao código de aprendizagem a fim de fornecer aos componentes de aprendizagem as informações contextuais. Isso dificulta a distinção entre o protocolo de aprendizagem e outros concerns de agente envolvidos. A adição, remoção e modificação do concern de aprendizagem de um SMA são normalmente invasivas, difíceis de reverter. Como separar o concern de aprendizagem dos demais concerns de agência? As forças a seguir surgem desse problema:

- A manutenção dos eventos e as estratégias de aprendizagem não devem afetar a funcionalidade básica do agente.
- Deve ser fácil reutilizar os protocolos básicos de aprendizagem para diferentes papéis e tipos de agente.
- As classes do kernel do agente não devem se misturar ao conhecimento específico à aprendizagem.

Solução. Usar aspectos para melhorar a separação do protocolo de aprendizagem. Os aspectos Learning (Figura 71) são usados para modularizar o protocolo de aprendizagem que afeta muitas partes de um agente de software. Eles separam o comportamento de aprendizagem das classes do agente, como tipos de agente, planos, papéis, crenças etc. Em outras palavras, os aspectos são usados não só para modularizar o centro do concern de aprendizagem, mas também para isolar todo o comportamento relacionado à coleta de informações e ao conhecimento específico à aprendizagem.

Usando os aspectos Learning, definimos quando e como o agente aprende. Eles conectam os pontos de execução (eventos) das diferentes classes do agente com as estratégias de aprendizagem correspondentes. Esses aspectos conseguem afetar alguns pontos de execução do agente – por exemplo chamadas de métodos nas classes do kernel do agente – a fim de alterar sua execução normal e disparar um componente de aprendizagem. Eles monitoram esses pontos de execução (eventos) a fim de identificar quando um processo de aprendizagem deve ser disparado. Esses eventos incluem a alteração de um elemento do conhecimento, execução de ações em planos, papéis, tipos de agente ou ainda alguma exceção levantada. As classes auxiliares são usadas para implementar diferentes técnicas de aprendizagem.

Estrutura. O padrão Aprendizagem possui dois participantes principais e um participante cliente:

Principais Participantes:

- **Aspecto Learning**

- define o protocolo de aprendizagem.

- **Componente de aprendizagem**

- Implementa uma estratégia de aprendizagem específica.

Participante cliente:

- **Elemento do conhecimento**

- Oferece os eventos relevantes e informações contextuais para fins de aprendizagem – pode ser uma classe Belief, uma classe Plan, uma classe Agent ou um aspecto do agente (papel, interação, ...).

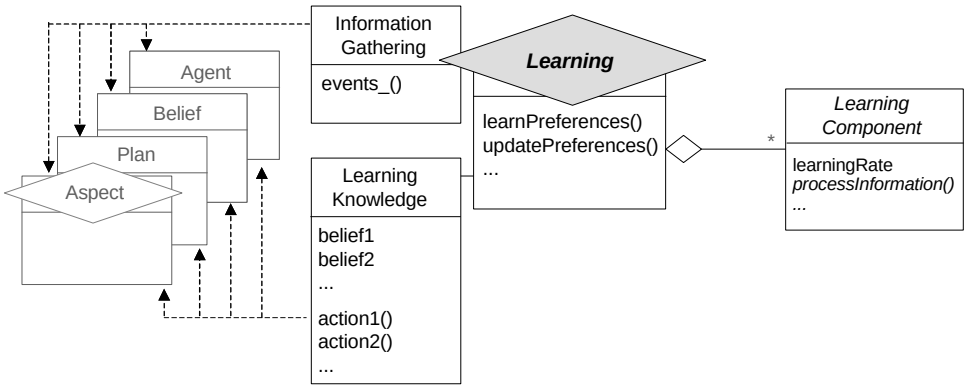


Figura 71. A visão estática do padrão Aprendizagem.

Na estrutura do padrão Aprendizagem (Figura 71), algumas partes são comuns a todas as instanciações potenciais do padrão, e outras partes são específicas a cada instanciação. As partes comuns são:

1. Os eventos genéricos em que o agente sempre aprende, por exemplo, quando uma mensagem é recebida.
2. O protocolo de aprendizagem geral:
    a. os eventos são detectados
    b. as informações contextuais são recebidas e
    c. os componentes de aprendizagem são chamados.
3. A lista de componentes de aprendizagem, ou seja, as referências a estratégias de aprendizagem mais sofisticadas.

As partes específicas são:

5. A especificação dos eventos específicos associados a um contexto específico em que o agente precisa aprender.
6. A coleta de informações específicas:
7. A configuração das estratégias de aprendizagem específicas usadas.

O propósito do aspecto Learning é tornar os agentes mais espertos. O aspecto Learning estende as classes do agente para introduzir o protocolo de aprendizagem. Ele possui três partes principais: o próprio aspecto e duas interfaces crosscutting. O seu propósito é tornar os agentes mais espertos. Além disso, ele estende as classes do agente para introduzir o protocolo de aprendizagem. Possui três partes principais: o próprio aspecto e duas interfaces crosscutting. O aspecto possui a lista de componentes de aprendizagem especializados e os métodos para atualizar o conhecimento do agente uma vez que novas conclusões são obtidas a partir dos componentes de aprendizagem. As interfaces crosscutting definem como o aspecto Learning afeta as diferentes classes de agentes de software. A interface InformationGathering define os join points que descrevem as informações e eventos relevantes que devem ser reunidos das classes do agente a fim de permitir o processo de aprendizagem. Ele contém os advices que chamam os métodos responsáveis pela implementação do comportamento de aprendizagem ou de um componente de aprendizagem específico. Os advices normalmente executam depois de execuções de ações de classes do agente, ações de classes do plano e outros aspectos associados ao agente (por exemplo, os aspectos de papéis – Seção 5.7). A interface LearningKnowledge introduz as diferentes crenças específicas à aprendizagem e as ações em diferentes classes do agente ou papéis com base em inter-type declarations.

A Figura 72 ilustra a instanciação de padrões do sistema Expert Committee. O aspecto Learning e seus subaspectos afetam os diferentes aspectos do agente e classes nesse sistema, cerca de 12 componentes. No entanto, a figura somente apresenta um conjunto parcial das classes afetadas pelos aspectos Learning. Ela mostra o aspecto Reviewer, a classe RevisionProposal, a classe UserAgent e a classe JudgementPlan. O aspecto Learning possui dois subaspectos: ChairLearning e ReviewerLearning; a

Figura 72 ilustra ReviewerLearning. Esse aspecto afeta a ação de julgamento de uma proposta, o método judgeProposal() na classe JudgementPlan, porque uma vez concluído o julgamento, as informações relacionadas a ele são usadas pelo aspecto Learning a fim de aprender sobre as preferências do usuário. O aspecto ReviewerLearning reúne as informações associadas ao julgamento da proposta e o componente de aprendizagem associado é chamado (a classe TDLearning nesse caso). O aspecto ReviewerLearning também intercepta os métodos no aspecto Reviewer e na classe UserAgent. A Figura 72 também ilustra como a interface LearningKnowledge do aspecto Learning modifica a estrutura da crença RevisionProposal. Ela introduz os atributos paperInterest e avaliação e os “setters” e “getters” associados.

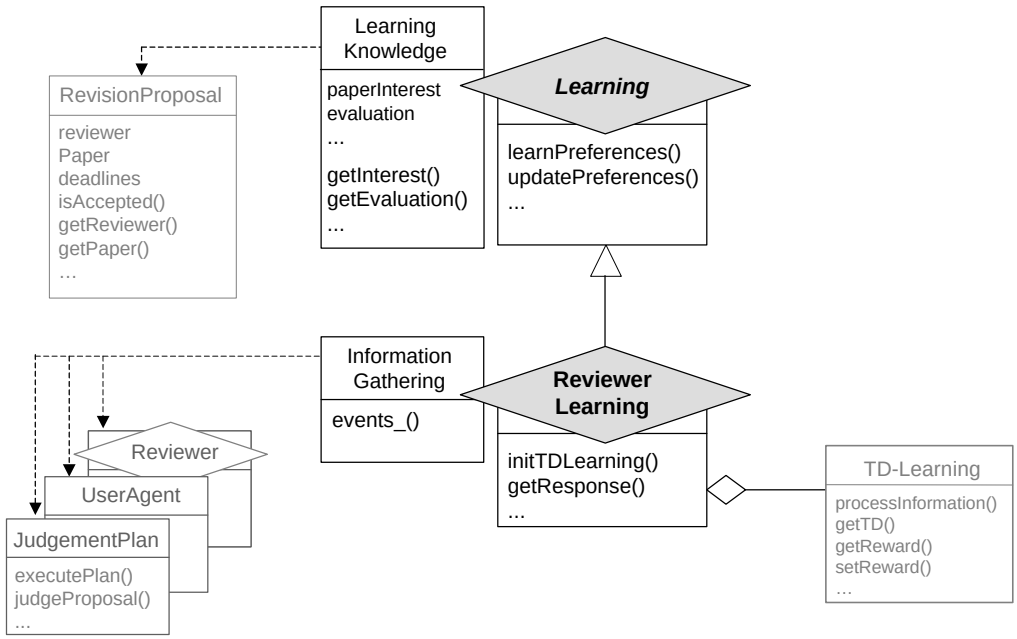


Figura 72. O padrão Aprendizagem para o agente de usuário do EC.

Observe que todo o código de aprendizagem foi removido das classes do agente e foi implementado separadamente em aspectos de aprendizagem associados, conforme já explicado. Esse tipo de aspecto parece ser um padrão orientado a aspectos genérico e comum, no qual os aspectos juntam a funcionalidade de suas classes associadas ao código do sistema original. De fato, o código de aprendizagem consiste em aspectos de aprendizagem e classes auxiliares ou interfaces devotadas a

implementar as estratégias de aprendizagem específicas. Quando esse código é entrelaçado ao código do sistema, ele afeta essencialmente as classes do agente; a comunicação entre os aspectos de aprendizagem afeta os objetos, como na classe RevisionProposal.

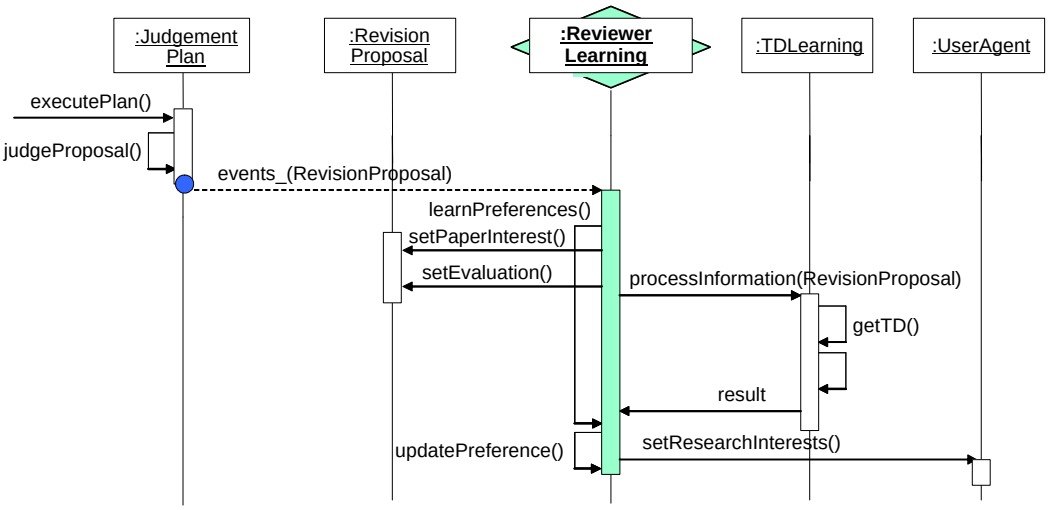


Figura 73. A visão dinâmica do aspecto ReviewerLearning.

Dinâmica. Os cenários a seguir descrevem o comportamento dinâmico do padrão Aprendizagem.

Cenário I – Reviewer Learning, ilustrado pela Figura 73, apresenta o comportamento do padrão quando o aspecto ReviewerLearning detecta que uma ação importante no plano do agente foi realizada e é necessária uma aprendizagem:

- O plano de julgamento é executado.
- As ações de julgamento são realizadas chamando o método judgeProposal()
- O aspecto ReviewerLearning detecta o final do julgamento interceptando essa execução de método.
- Esse aspecto reúne as informações necessárias do contexto do plano, ou seja o objeto RevisionProposal.
- O aspecto atualiza o objeto RevisionProposal de forma que chair possa aprender com base no julgamento do revisor – ele atualiza esse estado de

objeto chamando os métodos setPaperInterest() e setEvaluation(), introduzidos pelo aspecto Learning.

- O aspecto ReviewerLearning seleciona e chama os componentes de aprendizagem correspondentes, a classe TDLearning nesse caso, e fornece as informações contextuais.
- Ele executa seus algoritmos específicos e chega a uma conclusão, o que leva à atualização do kernel do agente, nesse exemplo a atualização de uma crença na classe UserAgent.

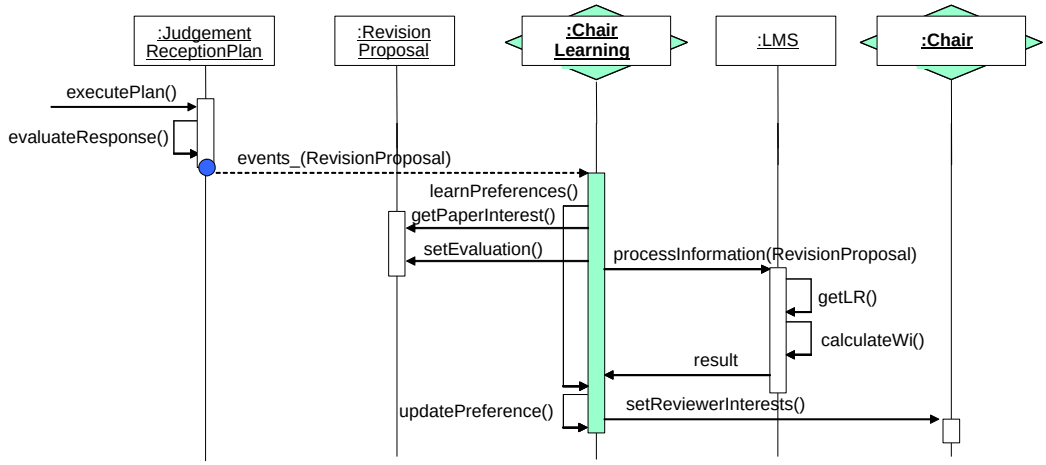


Figura 74. A visão dinâmica do aspecto ChairLearning.

Cenário II – Chair Learning, ilustrado pela Figura 74, apresenta o comportamento do padrão quando o aspecto ChairLearning detecta que uma ação importante no plano do agente foi realizada e é necessária uma aprendizagem:

- O plano para o recebimento do julgamento do revisor é executado.
- As ações para a avaliação do julgamento do revisor são realizadas chamando o método evaluateResponse().
- O aspecto ChairLearning detecta a execução dessa ação e intercepta o final da execução de método a fim de aprender com base na resposta do revisor.
- Esse aspecto reúne as informações necessárias do contexto do plano, ou seja, o objeto RevisionProposal.
- O aspecto reúne as informações necessárias do objeto RevisionProposal de forma que chair possa aprender com base no julgamento do revisor – ele pega

as informações chamando os métodos getPaperInterest() e getEvaluation(), introduzidos pelo aspecto Learning e atualizados pelo aspecto ReviewerLearning (Cenário I).

- O aspecto ChairLearning seleciona e chama os componentes de aprendizagem correspondentes, a classe LMS nesse caso, e fornece as informações contextuais.
- O aspecto executa seus algoritmos específicos e chega a uma conclusão, o que leva à atualização do conhecimento de chair, nesse exemplo a atualização de uma crença no aspecto de papéis Chair.

Os cenários apresentados anteriormente mostram o comportamento do padrão quando o processo é disparado pela execução de ações. Entretanto, os aspectos de aprendizagem podem interceptar as exceções que também contribuem para a aprendizagem do agente [39, 207, 172].

Conseqüências. O padrão Aprendizagem possui as seguintes conseqüências:

- *Reusabilidade*. O protocolo de aprendizagem básico é modularizado em um aspecto de aprendizagem genérico, que pode ser reutilizado e refinado em diferentes contextos. No exemplo anterior, os aspectos ChairLearning e ReviewerLearning reutilizam o protocolo de aprendizagem básico do aspecto Learning.
- *Melhor separação de concerns*. O protocolo de aprendizagem é totalmente separado dos demais concerns do agente, como a interação e os concerns básicos do agente. As classes e os aspectos associados aos demais concerns do agente não possuem código de aprendizagem.
- *Legibilidade e manutenibilidade*. O kernel do agente não é entrelaçado com chamadas de métodos responsáveis pela implementação da aprendizagem. Como conseqüência, melhora a legibilidade, que, por sua vez, melhora a manutenibilidade.
- *Menor replicação do código.* Conforme discutido anteriormente, o uso do padrão Observador introduz a replicação do código à medida que a

complexidade do agente aumenta. O padrão Aprendizagem oferece suporte ao isolamento do protocolo de aprendizagem em aspectos de aprendizagem, minimizando a replicação do código.

- *Transparência.* Os aspectos são usados para introduzir o comportamento da aprendizagem nas classes do agente de forma transparente. A descrição de quais classes do agente precisam ser afetadas está presente no aspecto, e essas classes monitoradas não são modificadas de forma intrusiva.
- *Facilidade de evolução*. Com a evolução do sistema multiagentes, novas classes do agente precisam ser monitoradas e disparam o processo de aprendizagem. Os desenvolvedores de SMAs só precisam de novos pointcuts no aspecto de aprendizagem a fim de implementar a nova funcionalidade necessária.

Variantes. *Aprendizagem reflexiva*. Esse variante é similar à solução orientada a aspectos apresentada aqui. A diferença está no uso dos metaobjetos como alternativa aos aspectos. No entanto, essa abordagem requer um protocolo metaobjeto [165] que normalmente introduz alterações na máquina virtual. Além disso, as soluções reflexivas não oferecem suporte à composição dos metaobjetos de aprendizagem com outros concerns. Como a complexidade do agente aumenta, bons mecanismos de composição são essenciais para a manutenibilidade e reusabilidade do sistema.

Aprendizagem indireta. A solução básica do padrão Aprendizagem considera o treinamento indireto para a aquisição de conhecimento [172, 207]; não trata de treinamento direto [172, 207]. Contudo, o variante da Aprendizagem indireta pode incluir uma classe adicional TrainingExperience para tratar da aprendizagem indireta.

Usos conhecidos. O framework Brainstorm [7, 8] implementa o variante de aprendizagem reflexiva. O Portalware e o Expert Committee implementaram o padrão Aprendizagem. O sistema Portalware implementa uma versão simplificada do padrão.

Padrões relacionados. O padrão Aprendizagem está relacionado ao padrão Kernel do Agente e ao padrão Papel porque o aspecto de aprendizagem aprende com base

nas ações do agente e nas ações do papel. Além disso, ele influencia as decisões e os comportamentos proativos que são executados pelo padrão Autonomia. O padrão Aprendizagem também pode ser usado juntamente com o padrão Adaptação, uma vez que o processo de aprendizagem pode ser disparado devido à adaptação do conhecimento.

O padrão Aprendizagem contém a implementação orientada a aspectos do padrão Observador [115]. O padrão Estratégia pode ser usado para implementar diferentes estratégias de aprendizagem, conforme apresentado anteriormente [137]. O padrão Façade pode ser usado para fornecer uma interface única para as estratégias de aprendizagem, independente da técnica usada. Finalmente, a implementação do padrão Adaptação (vide a seguir) usa alguns idiomas AspectJ [114], como *Template Advice*, *Composite Pointcut* e *Advice Method*.

5.10
Questões de implementação e implantação

Os padrões de projeto orientado a aspectos foram usados na implementação dos sistemas EC e Portalware. A implementação do EC baseou-se na versão 1.3 da linguagem AspectJ. A implementação do Portalware baseou-se na versão 0.8 (Seção 4.5). Além disso, os padrões de agência foram naturalmente incorporados na implementação de um sistema de gerenciamento de tráfego (Seção 7.1). Ela também usou o framework JADE [20] e uma arquitetura blackboard [155] para oferecer suporte à comunicação entre agentes. A integração dos padrões propostos com essas infra-estruturas ocorreu de forma direta. O Apêndice I apresenta em detalhes as questões de implementação e uma amostra de código.

Os aspectos Adaptation e Autonomy possuem pointcuts definidos para o mesmo join point: as execuções do método receiveMsg(). O construto de AspectJ declare precedence foi usado para especificar a ordem de execução entre esses aspectos (Apêndice I). Quanto ao concern de interação, havia vários join points nos quais as mensagens deveriam ser enviadas para outros agentes. Esses join points incluíam métodos em classes do plano e aspectos de papéis. A declaração de todos os métodos no pointcut outgoingMsg leva muito tempo. A fim de facilitar a especificação de

pointcuts, esses métodos foram nomeados com o prefixo “prepare”. A definição dos pointcuts usou um simples wildcard prepare* para capturar todos esses métodos.

A tradução dos padrões do projeto orientado a aspectos para o código AspectJ é quase direta, uma vez que sua representação baseia-se amplamente na terminologia POA. Entretanto, os padrões também podem ser implementados usando outros frameworks orientados a aspectos, como AspectWerkz e JBoss. Apesar de esses frameworks oferecerem suporte ao processo de combinação dinâmico, eles incorporam construtos similares à linguagem AspectJ. Hyper/J possui alguns construtos diferentes e não possui a dicotomia base-aspecto (Seção 2.2.2). No entanto, a Seção 7.3 mostra que os padrões propostos também podem ser implementados em Hyper/J usando um conjunto de heurísticas.

5.11 Discussão e trabalhos relacionados

Não é fácil entender as relações entre agentes e objetos a partir do ponto de vista da engenharia de software. Como conseqüência, é difícil promover a separação dos concerns do agente com base no paradigma de objetos. Embora haja ligações inegáveis entre agentes e objetos (Capítulo 3), poucos trabalhos de pesquisa na comunidade de padrões tentaram esclarecer essas relações. Apesar de alguns padrões isolados já terem sido propostos para agentes de software [10, 117, 250], eles são em boa parte de alto nível e não tratam da separação de concerns de agência nos níveis de implementação e projeto. Eles negligenciaram os desafios associados à interação entre propriedades do agente e as abstrações orientadas a objetos.

Nossa linguagem de padrões é um dos primeiros esforços nessa direção uma vez que: (i) demonstra como os padrões orientados a objetos não conseguem tratar da separação de concerns do agente e (ii) orienta a engenharia de software ao adicionar as propriedades de agência aos objetos de forma elegante e com base em abstrações orientadas a aspectos. Kendall et al [137] propõem algumas soluções orientadas a objetos para o projeto e a implementação dos concerns do agente com base em padrões de projeto conhecidos. Este capítulo ilustra muitos padrões de Kendall e os problemas associados. Em geral, as soluções orientadas a objetos não oferecem

suporte à separação de concerns, aumentam a esquizofrenia do agente e incorporam a replicação do código.

Kendall [136] descreve a aplicação da programação orientada a aspectos para implementar modelos de papéis. Essa abordagem é usada para representar os diferentes papéis que um agente pode exercer durante seu ciclo de vida. De fato, o padrão Papel apresentado neste capítulo segue algumas de suas diretrizes [136]. Entretanto, a solução de Kendall requer que os clientes do agente conheçam os aspectos de papéis; aumentando, então, o acoplamento do sistema. Ademais, seu trabalho não trata das relações entre papéis e outras propriedades do agente, que são a principal origem da complexidade do agente. Nesse sentido, esse capítulo apresenta uma linguagem de padrões unificadora para tratar de papéis e de outros concerns do agente e seus inter-relacionamentos.

A linguagem de padrões apresenta um padrão de projeto para a estruturação de papéis, mas a linguagem não incorpora um padrão para o concern de colaboração. É porque há alguns protocolos de colaboração que requerem padrões de projeto individuais para cada protocolo. Dessa forma, o concern de colaboração requer uma linguagem de padrões adicional. De fato, há alguns padrões para determinadas estratégias de coordenação e colaboração [117, 154]. Conforme mencionado anteriormente, o padrão Papel proposto pode ser usado juntamente com esses padrões existentes.

5.11.1 Vantagens e desvantagens

O uso de abstrações orientadas a objetos para o projeto do agente normalmente leva a problemas de crosscutting concerns e replicação de código (Seção 3.6). Os padrões propostos apresentam soluções orientadas a aspectos de forma que determinados concerns do agente entrelaçados em projeto orientado a objetos possam ser desentrelaçados. Os padrões melhoram a modularidade do sistema uma vez que os aspectos localizam toda a definição dos concerns do agente.

A melhora vem primariamente da modularização da implementação do protocolo associada às propriedades do agente. Por exemplo, o padrão Adaptação

encapsula o protocolo de adaptação, e o padrão Aprendizagem captura o protocolo de aprendizagem. Isso se reflete diretamente no fato de a implementação do protocolo ser textualmente localizada nos aspectos abstratos. Para conseguir isso, é preciso remover as dependências no nível do código das classes participantes para a implementação dos protocolos nos aspectos.

Essa separação de concerns permite que as aplicações específicas reutilizem o comportamento comum associado às propriedades do agente. Esse comportamento comum é fornecido pelos aspectos abstratos. Os desenvolvedores podem se concentrar na funcionalidade dependente da aplicação. Os aspectos abstratos implementam os comportamentos comuns, e os aspectos concretos especializam esses comportamentos para os tipos de agente e papéis. Além disso, apesar de os padrões orientados a aspectos serem apresentados como parte de uma linguagem de padrões unificadora, cada um dos padrões pode ser usado individualmente em aplicações de SMAs.

A transparência é uma propriedade essencial dos aspectos (Seção 2.2.2), que pode ser usada como uma medida informal que indica a utilidade de sistemas orientados a aspectos. A propriedade da transparência melhora a separação dos concerns no desenvolvimento de softwares e simplifica a análise, o projeto e a implementação das classes. Sistemas POA melhores devem ter maior transparência [67]; no entanto, a transparência total é um objetivo difícil de se alcançar, conforme discutido na próxima seção.

5.11.2 Lições aprendidas

Esta seção descreve algumas lições aprendidas a partir da experiência do uso dos padrões de projeto para o desenvolvimento do Portalware (Seção 4.1) e Expert Committee (Seção 5.1).

Exposição de Join Points

É muito provável que várias propriedades do agente em um SMA não tenham sido desenvolvidas desde o início como aspectos, conforme descrito nos padrões de

projeto. Em vez disso, muitos crosscutting concerns surgirão à medida que um SMA evolui. Um framework de avaliação pode ajudar na detecção de crosscutting concerns (Capítulo 6). Capturar esses concerns como aspectos, às vezes, requer a reestruturação das classes e dos métodos para expor os join points adequados. Por exemplo, extraímos o código dos métodos existentes de uma classe de plano para um novo método para expor um join point no nível do método de forma que o aspecto de papéis possa interceptá-lo. Nesses casos, a transparência do aspecto não é total. A *Intimidade* é definida como um esforço adicional necessário para preparar as classes e os métodos para a incorporação de aspectos ao sistema [71]. As ferramentas para ajudar a refatoração facilitariam a introdução de aspectos em um sistema existente.

Gerenciamento da complexidade do aspecto

Na implementação dos padrões, o autor observou que é mais fácil construir um sistema orientado a aspectos quando a interface entre aspectos e classes é estreita e unidirecional. *Unidirecional* significa que o código do aspecto se refere às classes, mas não vice-versa. De fato, no centro dos padrões propostos está a noção da estruturação de crosscutting concerns separadamente dos concerns do agente "primários", usando aspectos que não podem ser referenciados de volta pelos objetos. Estreito significa que o código do aspecto possui um efeito bem definido em pontos particulares nesse código. Lippert e Lopes fizeram observações similares com base em um estudo feito usando AspectJ para capturar os construtos de tratamento de exceção [161].

Aspectos como "Conectores" entre hierarquias de classes

Muitos padrões de projeto possuem uma estrutura comum: o código do aspecto forma a conexão entre duas estruturas orientadas a objetos. Por exemplo, os aspectos Learning são uma conexão entre a hierarquia de tipos de agente e a hierarquia de estratégias de aprendizagem. A estrutura de projeto é benéfica porque permite expressar a funcionalidade básica do agente na própria estrutura do objeto e usar um aspecto para injetar essa propriedade do agente na funcionalidade básica de forma transparente. Esse estilo deve ser melhor investigado como uma diretriz POA geral para tratar de outros crosscutting concerns.

Geração de código

A implementação dos padrões de projeto envolve algumas tarefas que demandam muito tempo na definição dos aspectos do agente, como uma descrição extensiva de pointcuts. Devem ser desenvolvidas ferramentas para maximizar a geração automática de pointcuts e realizar essas tarefas trabalhosas. Estamos desenvolvendo uma abordagem gerativa [55] que oferece suporte à geração de código dos padrões propostos [147]. O método arquitetural proposto e a linguagem de padrões proposta têm o suporte de diversas ferramentas e assistentes que automatizam a geração de código em AspectJ.

5.12 Resumo

Como as propriedades do agente estão sendo usadas hoje em dia em diferentes aplicações de software, os requisitos das aplicações baseadas em SMAs variam significativamente. Portanto, é importante que o projetista da aplicação configure as propriedades do agente para se ajustar às necessidades da mesma. Este capítulo apresentou uma linguagem orientada a aspectos para padrões de projeto orientado a aspectos que desacoplam propriedades de agência da funcionalidade de agente básica. As soluções orientadas a aspectos minimizam a esquizofrenia do agente, reduzem a replicação de código, aumentam a modularidade e a extensibilidade, e melhoram a reutilização dos componentes do agente.

Os padrões de projeto apresentados neste capítulo refinam os componentes da arquitetura do agente orientado a aspectos (Seção 4.3.2), fornecendo soluções de projeto detalhadas. Esses padrões seguem a estrutura geral da arquitetura orientada a aspecto e as diretrizes do método arquitetural (Capítulo 4). Os concerns identificados e modularizados no nível arquitetural são mapeados no nível de projeto detalhado usando abstrações orientadas a aspectos. Dessa forma, a separação alcançada no nível arquitetural é preservada no projeto detalhado.