4
O framework *fGrupos*

4.1
Objetivo do framework *f*Grupos

O principal objetivo do *f*Grupos é formar grupos de pessoas com interesse comum, e permitir que a interação entre os usuários também seja uma estratégia no processo de formação de grupos. O domínio desse *framework* se aplica a qualquer sistema *online* que necessite do processo de formação de grupos em prol da execução de uma tarefa comum, seja ela: o aprendizado colaborativo, atividades de voluntariado em grupo, transações de compras em grupo, agrupamento de pesquisadores interessados e capacitados para um projeto ou simplesmente a troca de informações entre pessoas que possuem o mesmo interesse ou desejam apenas se conhecer para a realização de um objetivo comum.

Assim, no âmbito comercial, uma organização poderá obter vantagens oferecendo um serviço que possibilite redução de custos aos clientes, possibilitando um aumento de receita para a organização e conseqüentemente maiores margens de lucro. Por outro lado, universidades, organizações sem fins lucrativos e comunidades virtuais também podem obter vantagens em atividades que necessitem reunir pessoas com competências e interesses em comum, pois os procedimentos necessários para a reunião destas pessoas serão automatizados pelo sistema.

4.2
Descrição Geral do framework fGrupos

Conforme mencionado anteriormente, o *f*Grupos oferece o serviço de formar grupos de pessoas com interesse comum, dentro de qualquer domínio da aplicação, e gerar um relatório, em qualquer formato de dados, referente ao grupo formado. Desta forma, o relatório pode ter alguma finalidade posterior, podendo

ser utilizada como dado de entrada para algum outro sistema que necessite de um grupo formado dentro do seu domínio de aplicação. Assim, o *framework* é flexível o suficiente, através de *hot spots*, para suportar o uso de quaisquer: (i) base de dados (relacional, semântica, orientada a objetos); (ii) heurística de definição de perfil do usuário; (iii) heurística de definição de perfil do grupo; (iv) estratégia de formação de grupos; (v) algoritmo de *matching* para a verificação dos perfis dos usuários associados ao perfil do grupo; (vi) algoritmo de encerramento do processo de busca por um grupo sugestivo; e, (vii) formato de apresentação do grupo formado (HTML , XML, TXT, e outros).

A flexibilidade no uso de qualquer base de dados é importante, pois a reutilização do *framework* torna-se mais abrangente, podendo ser aplicado em qualquer tipo de dados, seja ele semântico, relacional ou orientado a objeto.

A flexibilidade no uso de quaisquer heurísticas de definição de perfil do usuário e do grupo, qualquer estratégia de formação de grupos, e quaisquer algoritmos de *matching* e de encerramento do processo de busca por um grupo sugestivo é fundamental para que o *framework* seja reutilizado em qualquer aplicação pertencente a um domínio coberto por ele. Desta forma, cada aplicação poderá utilizar estratégias mais ajustadas com as características essenciais do seu domínio, tendo em vista apenas o tipo de dados utilizado e a padronização do mesmo dentro do processo de implementação dos *hot spots* do *framework*.

A flexibilidade no formato de apresentação do grupo formado é importante, pois o relatório de saída também compõe a finalidade do *framework*. Assim, os dados gerados podem ser utilizados por outras aplicações que necessitem de um grupo de interesse formado.

Para que o *f*Grupos fosse reutilizado no desenvolvimento de quaisquer sistemas de formação de grupos de interesse que precisassem ou não da interação entre os usuários para o processo de formação de grupos, a arquitetura do *framework* concentrou os *frozen spots* nos serviços oferecidos ao usuário (cadastrar-se, efetuar *login*, cadastrar perfil do grupo a ser formado, fazer *download* dos grupos formados, receber mensagem enviada por outro usuário e, efetuar *logout*) e, no processo de formação do grupo de interesse, que consiste da lógica de negócio do Sistema Multi-Agente definido.

Além disso, para que o *f*Grupos fosse reutilizado no desenvolvimento de sistemas monousuário, o *framework* forneceu uma funcionalidade, configurada no ato da sua instanciação, que consiste na obtenção de usuários cadastrados na base de dados e em suas ativações no ambiente através da criação e associação de agentes pessoais aos mesmos. Assim, o agente pessoal do usuário *online*, que entrou no sistema configurado conforme descrito acima, poderá formar grupos de interesse sem a necessidade da existência de outros usuários *online* ativos no ambiente.

Para fins de prova de conceito, foram realizadas três instanciações do *framework*:

- Um sistema de formação de grupos de pessoas com interesse científico comum. Esse sistema se aplica ao domínio de Projetos de Pesquisa.
- Um sistema de formação de grupos de voluntários. Esse sistema se aplica ao domínio de Ações de Voluntariado.
- Um sistema de formação de grupos de estudo. Esse sistema se aplica ao domínio de aprendizado colaborativo em ambientes de ensino.

A seguir apresenta-se uma visão geral do *framework*.

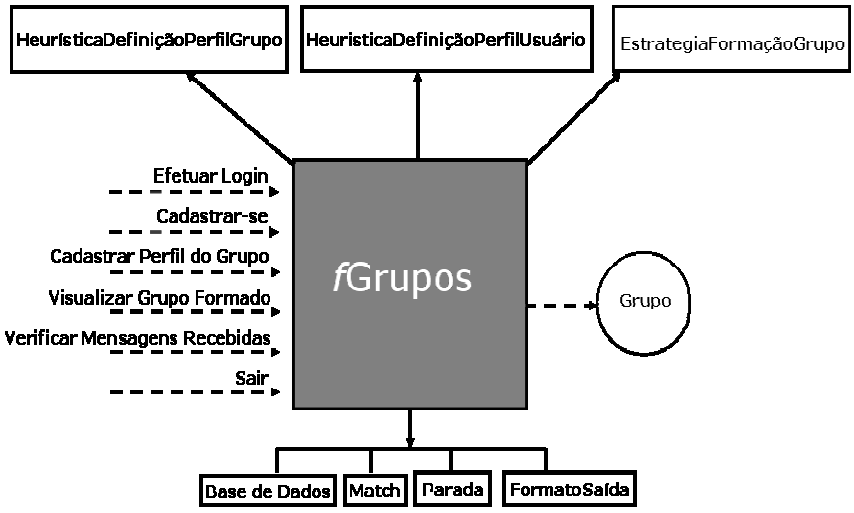

Figura 1: Visão geral do framework.

Na figura 1, o retângulo acinzentado representa o *framework*, os retângulos superiores e inferiores representam os pontos de flexibilização do *framework* (classes abstratas que devem ser implementadas no ato de sua instanciação), as

setas pontilhadas do lado esquerdo do retângulo acinzentado representam os serviços oferecidos ao usuário, e o círculo do lado direito do retângulo acinzentado representa a saída do *framework* (grupos formados).

De acordo com a figura 1, o *framework f*Grupos está dividido em duas funcionalidades distintas e complementares: (i) os serviços de interação com o usuário, oferecidos por uma instância genérica do *framework*; e, (ii) o SMA, responsável pelo processo de formação de grupos, que é integrante principal do núcleo do sistema juntamente com as classes concretas (CommunityManager e Group) e as classes abstratas que definem os pontos de flexibilização do *framework*.

4.2.1
Serviços de interação com o usuário

Como o nicho de mercado das aplicações que utilizam o *framework* está voltado para a Internet, o *fGrupos* possui uma interface Web acoplada, que oferece alguns serviços de acesso e interação ao ambiente. Os serviços oferecidos ao usuário são:

- Efetuar *login* no ambiente;
- Realizar seu cadastro no ambiente;
- Cadastrar o perfil de um grupo de interesse para a sua formação;
- Visualizar o resultado dos grupos solicitados para a formação;
- Verificar mensagens recebidas de outros usuários ativos no ambiente, caso seja utilizado este recurso disponível quando o processo de formação de grupos de interesse é interativo; e,
- Sair do ambiente encerrando a participação do usuário do processo de formação de grupos.

Assim, a instanciação do *framework* está restrita apenas à implementação de algumas classes abstratas e seus métodos abstratos, que são seus pontos de flexibilização. Entre eles, alguns métodos merecem maior atenção, pois a sua implementação deve seguir algumas regras definidas para que a interface Web funcione (descritas na seção 4.4).

4.2.2
Sistema Multi-Agente do *framework fGrupos*

O *f*Grupos foi desenvolvido baseado no conceito de Sistemas Multi-Agentes. Seguindo uma arquitetura BDI (Belief-Desire-Intention), o SMA desenvolvido para o *framework* utiliza agentes de software (pessoais e do grupo) de informação inteligentes e interativos para o processo de formação de grupos de interesse. A comunicação entre os agentes de software é do tipo direta e feita através do uso da linguagem ACL. A sua coordenação é verificada através do uso da estratégia de *matchmaking*. Desta forma, foi utilizado o ferramental Jade para o desenvolvimento do SMA.

A adoção do ferramental Jade se deve à sua compatibilidade com a linguagem de programação Java e também aos serviços de comunicação entre agentes oferecidos por ele.

Para o processo de formação de grupos de interesse, foi adotado o uso de dois tipos de agentes de informação inteligentes e interativos: agentes pessoais e agentes do grupo (*matchmakers*).

a) Agentes Pessoais.

Quando o usuário entra no ambiente, um agente pessoal é criado e adicionado na lista de agentes ativos da classe gerenciadora da comunidade. Sua finalidade é representá-lo dentro do processo de formação de grupos. Eles são os responsáveis por definir o perfil do usuário, seguindo uma heurística de definição de perfil do usuário, e comunicar-se com outros agentes de software ativos no ambiente que o requisitam ou que são requisitados por ele para uma possível formação de um grupo de interesse. O agente pessoal é encerrado somente após o *logout* do usuário no ambiente.

b) Agentes do Grupo.

Quando um usuário ativo no ambiente requisita a formação de um grupo de interesse e fornece o perfil do grupo, seguindo uma heurística de definição de perfil do grupo, um agente de software é criado. Sua finalidade é representar o grupo de interesse a ser formado, assumindo o papel de *matchmaker*. Para iniciar a formação do grupo requisitado, o agente de grupo solicita à classe gerenciadora da comunidade os agentes representantes dos usuários ativos no ambiente. Assim, através do uso de uma estratégia de *match* e um algoritmo de parada, o agente do

grupo seleciona aqueles que possuem características relacionadas com o perfil do grupo a ser formado. Em seguida, ele envia ao agente pessoal requisitante do grupo uma lista contendo os agentes pessoais ativos selecionados. Através desta lista, esse agente pessoal se comunica com os outros agentes pessoais integrantes da mesma, para a conclusão do processo de formação do grupo de interesse. Depois de finalizada a comunicação entre o agente pessoal do usuário que requisitou o grupo e os agentes integrantes da lista de agentes, o agente pessoal envia o grupo formado ao agente do grupo. Em seguida, o agente do grupo adiciona o grupo formado na classe gerenciadora da comunidade, que mantém uma lista de grupos formados requisitados por cada agente pessoal ativo no ambiente. Finalmente, o agente do grupo é finalizado.

4.3
Arquitetura do *framework f*Grupos

O *framework* foi desenvolvido utilizando o paradigma de orientação a objetos e a linguagem Java foi utilizada para a sua implementação. No entanto, o sucesso para o desenvolvimento de um *framework* com tecnologia orientada a objetos está fortemente ligado à arquitetura utilizada. A construção de uma arquitetura compreensiva, de qualidade e reutilizável pode ser uma tarefa árdua se não for atribuída alguma técnica para produção de projetos de software. Dentre elas, destaca-se a técnica de utilização de soluções típicas já encontradas para problemas-padrão. Essas soluções-padrão são conhecidas como padrões de projeto [44]. Assim, a arquitetura do *framework f*Grupos foi modelada baseada na utilização da alguns padrões de projetos, que serão apresentados no decorrer desta seção.

O *f*Grupos utiliza uma arquitetura multicamada, baseada no padrão de modelo MVC (Model View Controller) [27], que funciona de forma separada e inteligente levando em consideração a estrutura e a semântica de cada função a ser exercida. O objetivo é separar interface do usuário, lógica de negócio e dados da aplicação. Desta forma, a arquitetura do *framework* está dividida em três camadas e é representada através do uso de pacotes (Figura 2): a camada do cliente, a camada do modelo e a camada do controle.

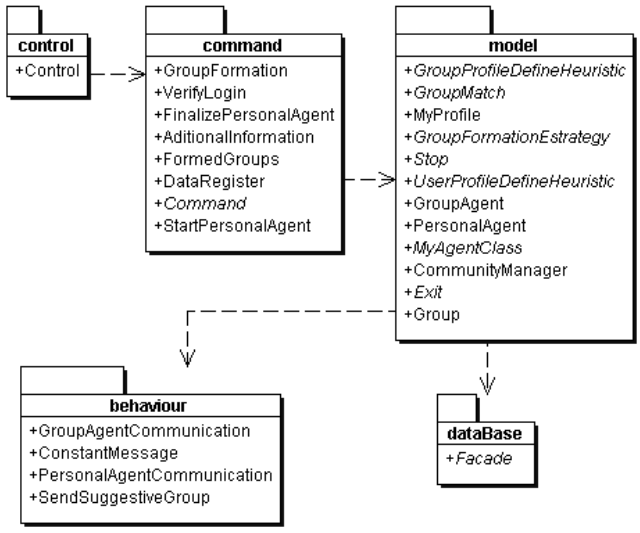

Figura 2: Diagrama de Pacote do framework fGrupos.

a) *Camada do cliente*

A camada do cliente fornece a interface do usuário. Em uma aplicação Web, sua composição consiste de páginas Web.

b) *Camada do controle*

A camada do controle é responsável pela conexão da camada do cliente à camada do modelo. Na arquitetura do *f*Grupos, sua representação ocorre nos pacotes control e command, apresentados abaixo.

A seguinte camada foi modelada baseada no padrão comportamental command [27]. O objetivo é associar uma ação (funcionalidade da classe Control) a diferentes objetos (objetos das classes do pacote command) através de uma interface conhecida (classe abstrata Command).

O diagrama de classe do pacote control na figura 3 é representado pela classe Control. Essa classe é responsável pelo recebimento das requisições do cliente e pela execução da respectiva classe do pacote command, a qual está associada ao serviço requerido. Sua visualização é apresentada na figura 4.

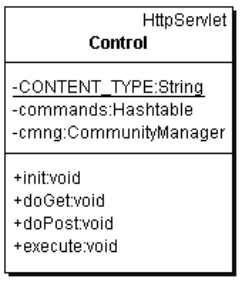

Figura 3: Diagrama de Classe do pacote control.

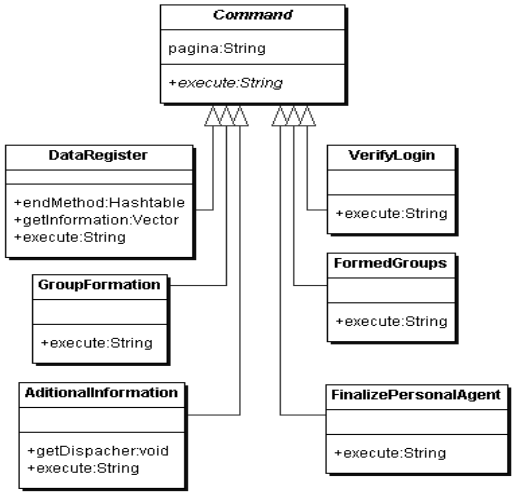

Figura 4: Diagrama de Classe do pacote command.

O diagrama de classe do pacote command na figura 4 é representado pelas classes VerifyLogin, DataRegister, GroupFormation, FormedGroups, AditionalInformation e FinalizePersonalAgent. As classes são estruturadas através da aplicação do padrão de projeto comportamental *command*, onde as mesmas são utilizadas para efetuar a conexão entre a camada do cliente e a camada do modelo. Assim, quando um serviço é acionado pelo cliente através de páginas Web, o controle do *framework* redireciona a execução do serviço para uma das classes do pacote command.

c) *Camada do Modelo*

A camada do modelo é responsável pela lógica de negócio da aplicação e, fornece acesso direto aos dados. Na arquitetura do *f*Grupos, esta camada é representada pelos pacotes model, dataBase e behaviour. O diagrama de classe do pacote model na figura 5 é representado pelas classes concretas CommunityManager, Group, MyProfile, MyAgentClass, PersonalAgent, e

GroupAgent, e pelas classes abstratas UserProfileDefineHeuristic, GroupProfileDefineHeuristic, GroupFormationEstrategy, Stop, GroupMatch e Exit. Essas classes são responsáveis pela lógica de negócio do Sistema Multi-Agente do *framework*.

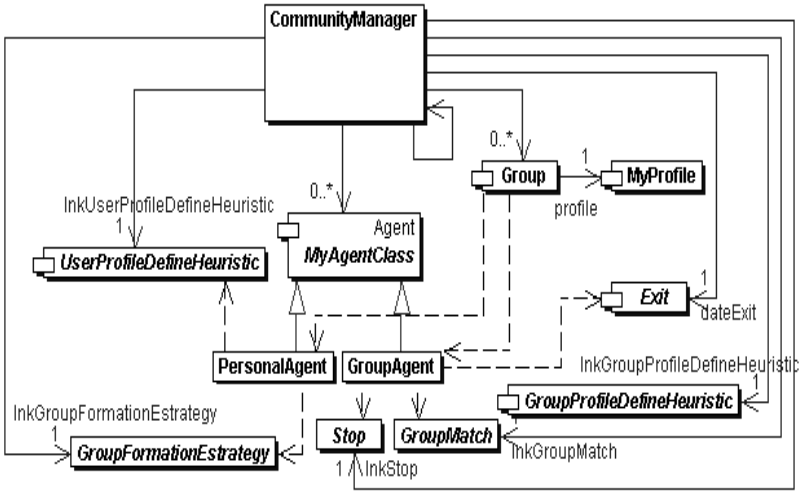

Figura 5: Diagrama de Classe do pacote model.

Os objetos da classe Group representam os grupos de interesse requisitados no ambiente, juntamente com as informações necessárias para as suas formações. Essas informações são compostas pelo perfil do grupo (objeto da classe MyProfile), pelo agente do grupo representante, pelo agente pessoal representante do usuário requisitante do grupo e, pela lista de agentes pessoais que constituem o grupo formado.

A classe abstrata MyAgentClass e as suas classes concretas PersonalAgent e GroupAgent representam, respectivamente, os agentes pessoais e de grupo, utilizados no processo de formação de grupos.

Finalmente, as classes abstratas UserProfileDefineHeuristic, GroupProfileDefineHeuristic, GroupFormationEstrategy, Stop, GroupMatch e Exit representam os pontos de flexibilização do *framework* e, devem ser implementadas no ato da instanciação do *f*Grupos. Para a flexibilização dos serviços oferecidos por essas classes, as suas estruturações ocorreram através da aplicação do padrão de projeto comportamental *Strategy* [27]. Assim, o seu uso

permitiu que uma família de algoritmos fosse utilizada de modo independente e seletivo.

O diagrama de classe do pacote dataBase na figura 6 é representado pela classe abstrata Facade. Seu objetivo é atuar como uma fachada [27] para a base de dados, fornecendo o acesso e a recuperação de informações da mesma.

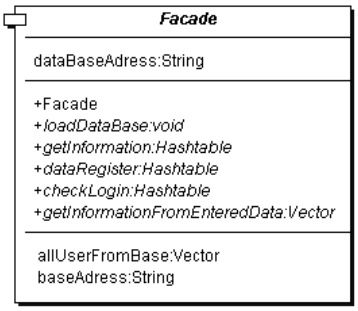

Figura 6: Diagrama de Classe do pacote dataBase.

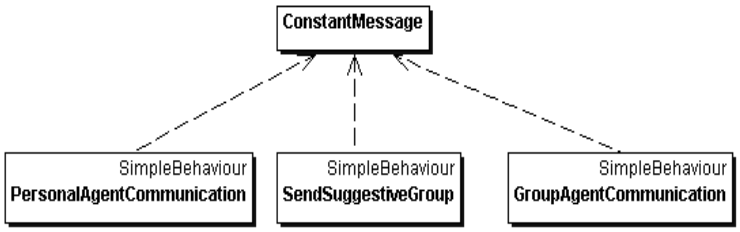

Figura 7: Diagrama de Classes do pacote behaviour.

O diagrama de classes do pacote behaviour na figura 7 é representado pelas classes concretas ConstantMessage, PersonalAgentCommunication, SendSuggestiveGroup e GroupAgentCommunication. Essas classes são responsáveis pela comunicação entre os agentes utilizados.

A divisão da arquitetura do *framework* em camadas é a chave para a independência entre os componentes e, como conseqüência, permitiu uma maior compreensão do projeto, facilidade de manutenção, eficiência e reutilização.

A seguir será apresentado o diagrama de casos de uso do *framework.* Através dele serão descritos: (i) o detalhamento dos serviços de interação com o usuário, fornecidos por uma instância genérica do *f*Grupos, centrando a explicação

nas conexões existentes entre a camada do cliente e a camada do modelo da aplicação e; (ii) o funcionamento interno do SMA.

4.3.1
Detalhamento dos serviços de interação com o usuário

A descrição detalhada dos serviços de interação com o usuário é interessante para a compreensão geral do funcionamento interno do *framework*. Assim, para facilitar a compreensão do fluxo de dados entre as classes do *framework*, o diagrama de casos de uso do *f*Grupos é apresentado a seguir.

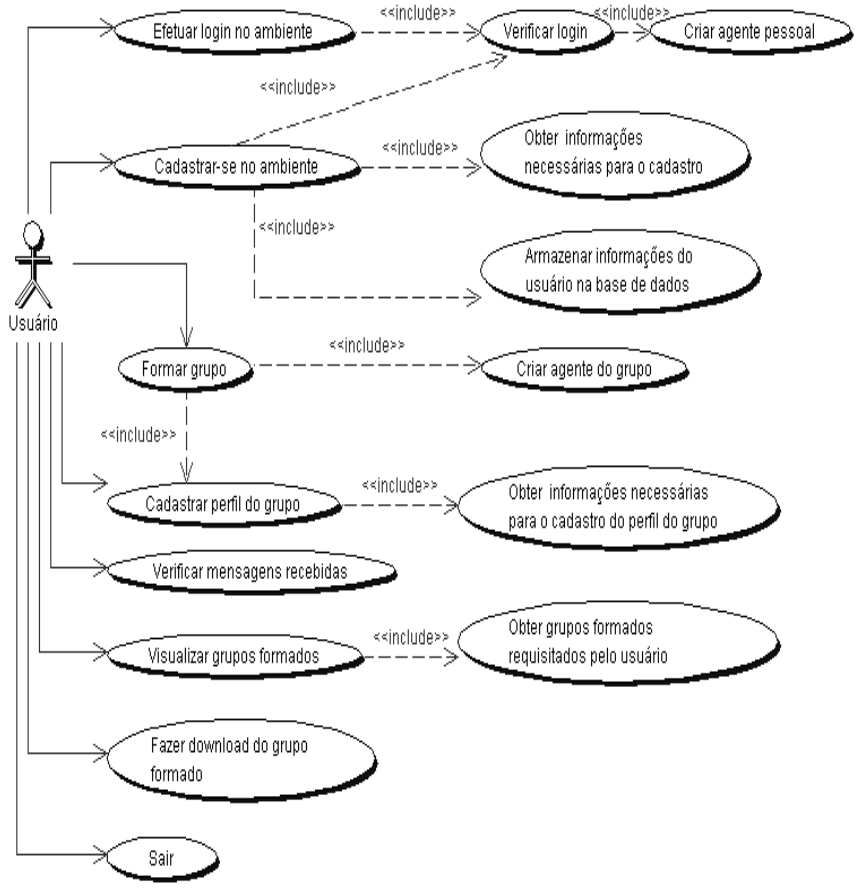

Figura 8: Diagrama use case do *framework f*Grupos.

A figura 8 apresenta o diagrama de casos de uso do *f*Grupos, onde é possível observar as interações entre o usuário e uma dada instância do *framework*. A seguir, serão descritas com maiores detalhes as interações que uma instância genérica do *framework* oferece.

a) Efetuar Login no ambiente.

Nesta interação, o usuário entra no sistema e inicia sua participação no ambiente através de um *login* (identificação do usuário no sistema) que é gerado no ato de seu cadastro. Para que a verificação do *login* seja realizada, o método abstrato checkLogin da classe abstrata Facade do pacote dataBase deve ser implementado.

Após entrar no ambiente, o sistema disponibiliza ao usuário os serviços de formar grupo, visualizar grupos formados, verificar mensagens recebidas por outros usuários que desejem se comunicar com ele, e sair do ambiente (efetuar *logout*).

Além disso, também após a sua entrada, um agente pessoal é criado para a definição do seu perfil a partir de informações pessoais encontradas na base de dados. A definição do perfil é feita seguindo uma heurística de definição de perfil de usuários. Para isso, o método abstrato getDefHeuristic da classe abstrata UserProfileDefineHeuristic do pacote model deve ser implementado.

Esse perfil é usado no auxílio do processo de formação de grupos. O agente pessoal se mantém ativo no ambiente enquanto o usuário também estiver ativo no sistema. Sua finalidade é representar o usuário, seja quando o seu proprietário desejar formar um grupo, ou quando ele estiver sendo avaliado por um agente representante de outro usuário que requisitou o processo de formação de um grupo de interesse.

A comunicação entre a camada do cliente e a camada do modelo do *framework* no processo de verificação de *login* e da criação do agente pessoal é intermediada através da classe VerifyLogin do pacote command.

b) Cadastrar-se no ambiente.

Nesta interação, o usuário cadastra-se no sistema e habilita sua possível entrada no ambiente. As informações necessárias para o cadastro foram flexibilizadas. Assim, elas variam de acordo com a instância do *framework* e são de responsabilidade do instanciador. Para isso, o método abstrato getInformationToUserRegister da classe abstrata UserProfileDefineHeuristic do pacote model deve ser implementado seguindo algumas regras necessárias para o funcionamento da interface Web (descrito na seção 4.4.1).

Alguns dados fornecidos para serem selecionados pelo usuário podem depender da seleção de dados anteriores. Neste caso, as informações necessárias para o cadastro do usuário são obtidas dinamicamente. Para isso, o método abstrato getInformationFromEnteredData da classe abstrata Facade do pacote dataBase deve ser implementado, conforme descrito na seção 4.4.1.

Após o fornecimento dos dados cadastrais do usuário as informações devem ser armazenadas na base de dados. No entanto, o *framework* flexibilizou o uso de qualquer base de dados. Desta forma, o método abstrato dataRegister da classe abstrata Facade do pacote dataBase deve ser implementado para que o armazenamento dos dados seja possível.

Após o seu registro, o usuário recebe uma identificação única, seja ela através do seu *login*, CPF ou nome de usuário. Em seguida, a etapa anterior é automaticamente iniciada.

A comunicação entre a camada do cliente e a camada do modelo do *framework* no cadastramento do usuário é intermediada através da classe DataRegister do pacote command.

c) Formar Grupo

Após entrar nesse processo interativo, o usuário deverá iniciar o processo de formação de grupo, cadastrando o perfil do grupo o qual ele deseja formar (ver procedimento descrito abaixo). Em seguida, o perfil do grupo é criado, baseado numa heurística de definição de perfil do grupo. Para isso, o método abstrato getGroupProfile da classe abstrata GroupProfileDefineHeuristic do pacote model deve ser implementado.

Após o cadastro do perfil do grupo, um objeto da classe Group do pacote model é gerado e, na seqüência, um agente do grupo é criado e associado a ele, com a finalidade de iniciar e gerenciar o processo de formação do grupo de interesse requerido (ver seção 4.3.2). Esse processo de criação de um objeto da classe Group e de um agente de grupo é realizado através do método createGroup da classe CommunityManager do pacote model.

No entanto, o processo de formação de grupo pode ser demorado e o usuário pode não esperar pelo seu término. Assim, convencionou-se em utilizar uma estratégia de lógica de negócio, a qual permite que o usuário inicie vários processos de formação de grupos de interesse.

A comunicação entre a camada do cliente e a camada do modelo do *framework* no processo de formação de grupos é intermediada através da classe GroupFormation do pacote command.

d) Cadastrar perfil do grupo

Nesta interação, o usuário cadastra o perfil do grupo que ele deseja formar. As informações necessárias para o cadastro do perfil do grupo variam de acordo com a instância do *framework*, sendo de responsabilidade do instanciador. Para isto, o método abstrato getInformationToGroupRegister da classe abstrata GroupProfileDefineHeuristic do pacote model deve ser implementado seguindo algumas regras necessárias para o funcionamento da interface Web, conforme descrito na seção 4.4.1.

Alguns dados fornecidos para serem selecionados pelo usuário para o cadastro do perfil do grupo podem depender da seleção de dados anteriores. Neste caso, as informações necessárias para o cadastro do perfil do grupo são obtidas dinamicamente. Para isso, o método abstrato getInformationFromEnteredData da classe abstrata Facade do pacote dataBase deve ser implementado, conforme descrito na seção 4.4.1.

Após o fornecimento dos dados cadastrais do perfil do grupo, o processo de formação de grupo é iniciado.

A comunicação entre a camada do cliente e a camada do modelo do *framework* no cadastramento do perfil do grupo é intermediada também através da classe DataRegister do pacote command.

e) Verificar Mensagens Recebidas

Conforme descrito no início deste capítulo, o *framework f*Grupos permite que a interação entre usuários também seja uma estratégia no processo de formação de grupos. Desta forma, foi disponibilizado um serviço que permite a um agente pessoal, representante de um usuário que requisitou o processo de formação de grupo, o envio de uma mensagem a um outro usuário ativo no ambiente, candidato a participar do grupo a ser formado. Essas mensagens são apresentadas através de formulários Web compostos por informações que o usuário receptor da mensagem deverá responder para que o processo de formação de grupo de interesse seja realizado. Assim, quando o *framework* for instanciado para fazer uso dessa interação como uma estratégia de formação de grupos, o

método abstrato getInformation da classe abstrata GroupFormationEstrategy do pacote model deverá ser implementado através da utilização de recursos disponíveis que serão descritos na seção 4.4.1.

Em virtude desse serviço, um usuário ativo no ambiente poderá verificar as mensagens recebidas por outros usuários que desejam a sua participação em um grupo de interesse. A obtenção dessas mensagens ocorre através da chamada do método getMessage da classe CommunityManager do pacote model, que é utilizado pela classe AditionalInformation do pacote command. Essa classe é responsável pela comunicação entre a camada do cliente e a camada do modelo do *framework* para a obtenção das mensagens recebidas e para o fornecimento de um aviso referente à chegada de uma nova mensagem.

Após receber um aviso, o usuário poderá visualizar a mensagem constada nele com apenas um clique. A visualização da mensagem ocorre através de um formulário o qual deverá ser preenchido ou respondido, com a finalidade de enviar a resposta ao remetente da mensagem. Esse último processo é gerenciado pela classe DataRegister do pacote command, que é responsável pela comunicação entre a camada do cliente e a camada do modelo do *framework* para o fornecimento das respostas recebidas ao remetente.

f) Fazer download do grupo formado

Conforme seção 3.5.5, o *framework* fornece uma interface que permite que o processo de formação de grupos seja apresentado ao usuário em duas etapas: a etapa do cadastro do perfil do grupo e a etapa da obtenção dos grupos formados requisitados por ele. Assim, o usuário pode iniciar o processo sem a necessidade de esperar pela finalização do mesmo. Em virtude disso, a formação de grupos de interesse poderá ser acionada quantas vezes o usuário precisar ou desejar.

Após o término do processo de formação de um grupo de interesse requisitado por um usuário, o agente do grupo aciona o método concreto createGroupExit da classe abstrata Exit do pacote model para que através dele, o grupo formado assuma o formato de saída definido na instanciação do *framework* e, posteriormente seja adicionado na lista de grupos formados requeridos pelo agente pessoal. Em seguida, o agente do grupo é finalizado. Desta forma, o usuário poderá, sempre que necessário, fazer *download* dos grupos formados que foram requisitados por ele. A obtenção dos grupos formados se deve através da

chamada do método getGroupsFormedFromPersonalAgent da classe CommunityManager do pacote model.

A comunicação entre a camada do cliente e a camada do modelo do *framework* no processo de *download* do grupo formado é intermediada através da classe FormedGroups do pacote command.

g) Sair do ambiente

Ao sair do ambiente, o agente pessoal criado através do *login* é finalizado. Quando isso ocorre, este agente é retirado da lista de agentes pessoais ativos no ambiente da gerenciadora da comunidade, juntamente com as informações referentes aos grupos de interesse formados requisitados pelo usuário representado por ele. Esse processo é efetuado através da chamada do método finalizePersonalAgent da classe CommunityManager do pacote model.

A comunicação entre a camada do cliente e a camada do modelo do *framework* no processo de sair do ambiente é intermediada através da classe FinalizePersonalAgent do pacote command.

4.3.2 Funcionamento interno do SMA

Conforme descrito na seção 4.2, o *framework f*Grupos está dividido em duas funcionalidades: os serviços de interação com o usuário; e o SMA responsável pelo processo de formação de grupos.

Os serviços de interação com o usuário (efetuar login, cadastrar-se, formar grupo, cadastrar perfil do grupo, visualizar grupos formados, sair) são disponibilizados e, quando necessário, são solicitados para que o processo de formação de grupos de interesse seja disponibilizado e então acionado.

Assim, depois do fornecimento do perfil do grupo a ser formado e da obtenção dos perfis dos usuários ativos no ambiente (efetuada pelos agentes pessoais), o processo de formação de grupos é iniciado baseando-se na comunicação entre os agentes de software. A comunicação entre os agentes pessoais e a comunicação entre os agentes pessoais e os agentes de grupo é realizada através da troca de mensagens entre os mesmos. Para isto foi utilizada uma linguagem de comunicação entre agentes denominada ACL, apresentada anteriormente, a qual define os tipos de mensagens trocadas entre os agentes. A

definição dos tipos de mensagens se verifica através do uso da classe ConstantMessage do pacote behaviour.

Perante as dificuldades encontradas para a construção direta do Sistema Multi-Agente e conseqüentemente do *framework*, decidiu-se utilizar a ferramenta Jade, apresentada anteriormente, que é um ambiente para o desenvolvimento de aplicações baseadas em agentes. A definição dos agentes de software e a interação entre os mesmos ocorrem através da adoção de comportamentos associados a cada um. Por isso, os agentes pessoais e os agentes de grupos possuem comportamentos diferenciados os quais foram pré-definidos durante a projeção da arquitetura do *framework*. Esses comportamentos são representados pelas classes GroupAgentCommunication, PersonalAgentCommunication e SendSuggestiveGroup do pacote behaviour.

A seguir serão apresentados os comportamentos dos agentes de software no processo de formação de grupos.

4.3.2.1 Agente Pessoal

Quando um agente pessoal é criado, o mesmo é adicionado na lista de agentes pessoais ativos no ambiente da gerenciadora da comunidade (CommunityManager), para em seguida, ser iniciado. A sua finalidade é definir o perfil do usuário a ser representado e continuar ativo no ambiente enquanto o mesmo não efetuar *logout*.

Durante o seu ciclo de vida, o agente pessoal assume um comportamento pré-definido que consiste em receber as mensagens enviadas por um agente e, de acordo com o tipo de mensagem, executar uma seqüência de ações para que o processo de formação de grupos seja realizado.

A comunicação estabelecida por um agente de grupo a um agente pessoal se deve somente através de dois fatores: o envio de um grupo sugestivo de agentes pessoais ativos no ambiente, que estão aptos a participarem do grupo de interesse requisitado; e, o recebimento de um grupo de interesse formado pela parte ou pelo todo dos integrantes do grupo sugestivo original.

A comunicação entre dois agentes pessoais é estabelecida somente em virtude de dois propósitos: (i) o envio de respostas a mensagens recebidas, e; (ii) o

recebimento de informações requeridas que são necessárias para avaliar a aptidão dos agentes pessoais integrantes do grupo sugestivo recebido pelo agente do grupo.

Um agente pessoal pode receber, durante seu ciclo de vida, apenas quatro tipos de mensagens: (i) recebimento do grupo sugestivo; (ii) requerimento de informação; (iii) recebimento de informação requerida, e; (iv) recebimento de alerta do *timeout*. O comportamento assumido associado a cada uma delas é descrito a seguir.

a) Recebimento do grupo sugestivo

Este tipo de mensagem ocorre quando o agente pessoal admite o recebimento de uma mensagem que foi enviada pelos agentes dos grupos que o mesmo requisitou. Ou seja, essa mensagem é recebida apenas quando o usuário representado pelo agente pessoal receptor requisitou a formação de um grupo. Desta forma, o corpo da mensagem é composto por uma lista de agentes pessoais que possuem alguma característica em comum com o perfil do grupo requisitado pelo usuário representado pelo agente pessoal receptor desta mensagem. Após a obtenção dessa lista, o agente pessoal receptor envia uma mensagem para cada agente integrante da lista, requisitando o envio de uma informação. Essa informação é utilizada no processo de formação de grupo. O tipo das informações trocadas entre os agentes pessoais é definido pela estratégia de formação de grupo utilizada na instância do *framework*.

b) Requerimento de Informação

Este tipo de mensagem ocorre quando o agente pessoal admite o recebimento de uma mensagem, enviada por um outro agente pessoal, definida como requerimento de informação. Essa mensagem pode ser recebida por qualquer agente pessoal que estiver ativo no ambiente, pois este está sujeito a ser membro de qualquer grupo sugestivo de qualquer agente de grupo. Essa mensagem requisita informações pessoais do usuário representado pelo agente pessoal receptor. Essas informações são definidas no ato da instanciação do *framework* através da estratégia de formação de grupo utilizada. Assim, o método abstrato getInformation da classe abstrata GroupFormationEstrategy do pacote model deve ser implementado e utilizado para a obtenção das informações requeridas. O mesmo retorna uma tabela *hash*, que armazena as informações que

devem ser trocadas entre os agentes pessoais. Em seguida, o agente pessoal responde a mensagem original, enviando no corpo da mensagem modificada a tabela *hash* obtida.

c) Recebimento de Informação Requerida

Um agente pessoal admite o recebimento de uma mensagem de resposta enviada por um outro agente pessoal. Essa mensagem é recebida apenas pelos agentes pessoais que inicialmente enviaram mensagens de requerimento de informação a outros agentes pessoais. Desta forma, o corpo desta mensagem resposta é composto por uma tabela *hash* que armazena as informações pessoais requisitadas na mensagem original. Após a obtenção dessas informações, o agente pessoal representante do usuário requisitante do grupo verifica se o agente pessoal remetente dessas informações está apto a participar do grupo de interesse em questão. Essa verificação é definida no ato da instanciação do *framework* através da estratégia de formação de grupo utilizada. Assim, o método abstrato getAccept da classe abstrata GroupFormationEstrategy do pacote model deve ser implementado e utilizado para a obtenção da verificação requerida. Essa verificação pode ser definida, por exemplo, através de um algoritmo de *matching* entre os perfis dos usuários representados pelos agentes pessoais envolvidos na mensagem (remetente e destinatário). O retorno dessa chamada de método é um valor lógico que determina a participação ou não de um usuário no grupo a ser formado.

Depois de receber a resposta de todas as mensagens de requerimento de informação enviadas aos agentes pessoais integrantes do grupo sugestivo recebido, o agente pessoal finaliza o processo de formação do grupo de interesse em questão e envia ao agente de grupo o grupo formado.

d) Recebimento de Alerta de TimeOut

Após ser enviada por um agente de grupo, o agente pessoal admite o recebimento de uma mensagem que é definida como recebimento de alerta de *timeout*. Essa mensagem é recebida apenas quando o usuário, representado pelo agente pessoal receptor, requisitou a formação de um grupo. Essa mensagem alerta ao agente pessoal que o tempo esperado pelo processo de formação do grupo de interesse coordenado por ele se esgotou. Assim, o agente pessoal deve encerrar sua comunicação com os agentes

pessoais integrantes do grupo sugestivo recebido. Tal estratégia foi adotada para evitar que um agente pessoal fique esperando pela resposta de um outro agente pessoal que esteve ativo no ambiente, mas que foi desativado antes que pudesse responder às mensagens recebidas.

4.3.2.2 Agente do Grupo

No ato da instanciação do *framework*, é criada uma heurística de definição de perfil do grupo que é utilizada no momento em que um usuário solicita a formação de um grupo de interesse. Ela fornece os dados necessários para a definição do perfil do grupo. Em seguida, um agente do grupo é criado para a coordenação do processo de formação do grupo de interesse requisitado. Finalmente, um objeto da classe Group do pacote model é gerado. Essa classe armazena informações referentes ao grupo representado por ela, as quais são necessárias para o processo de formação de grupos dentro do sistema. Assim, o perfil do grupo definido, o agente pessoal representante do usuário requisitante do grupo e o agente do grupo criado são associados ao objeto da classe Group gerado. Em seqüência, o mesmo é adicionado na gerenciadora da comunidade, através da sua inserção na lista de grupos requeridos por um agente pessoal ativo no ambiente.

Após a sua criação, o agente do grupo é iniciado com a finalidade de: (i) obter uma lista de agentes pessoais ativos no ambiente (grupo sugestivo), que possuem características comuns com o perfil do grupo requerido; (ii) enviar o grupo sugestivo ao agente pessoal que representa o usuário requerente do grupo, e; (iii) esperar pelo grupo formado que é enviado após o término da comunicação entre o agente pessoal, representante do usuário que requisitou o grupo, e os agentes pessoais integrantes do grupo sugestivo.

a) Obtenção do grupo sugestivo

A obtenção do grupo sugestivo é efetuada através da utilização de um algoritmo de *match* e de um algoritmo de parada durante o processo de obtenção do mesmo.

O agente do grupo inicia uma iteração para a obtenção do grupo sugestivo. O encerramento do processo iterativo é determinado pelo algoritmo de parada, verificado através da chamada do método abstrato getStop da classe abstrata Stop do pacote control.

A cada iteração, o agente do grupo obtém um agente pessoal ativo no ambiente através da chamada do método getPersonalAgentToGroupAgent da classe CommunityManager do pacote model. Em seguida, é feita uma análise comparativa entre o perfil atribuído ao agente pessoal retornado e o perfil do grupo requisitado. O objetivo é verificar se esse agente pode ser integrante do grupo sugestivo em formação. O resultado dessa comparação é determinado pelo algoritmo de *match,* verificado através da chamada do método abstrato getMatch da classe abstrata GroupMatch do pacote control.

b) Envio do grupo sugestivo

Em seguida, o agente do grupo efetua o comportamento de enviar o grupo sugestivo obtido para o agente pessoal representante do usuário que requisitou o processo de formação do grupo. Esse comportamento é verificado através da chamada do método construtor da classe SendSuggestiveGroup do pacote behaviour.

c) Espera pelo término do processo de formação do grupo

Finalmente, o agente do grupo ativa o seu comportamento de comunicação. Esse comportamento é verificado através da chamada do método construtor da classe GroupAgentCommunication do pacote behaviour, o qual define que o agente do grupo se mantém ativo no ambiente e deve esperar pelo recebimento de uma mensagem enviada pelo agente pessoal em resposta a mensagem original de envio de um grupo sugestivo. Essa mensagem de resposta contém o grupo formado. Em seguida, o agente do grupo faz a chamada do método concreto createGroupExit da classe abstrata Exit do pacote control para que, através dela, os dados referentes ao grupo formado sejam formatados de acordo com o formato de saída definido no momento da instanciação do *framework.* Estes dados são adicionados na lista de grupos formados que foram requisitados por um agente pessoal da gerenciadora da comunidade (CommunityManager). A formatação dos dados de saída é efetuada através da chamada do método abstrato formatParser da

classe abstrata Exit do pacote control. Somente após esse processo, o agente do grupo é finalizado.

No entanto, o agente pessoal admite o recebimento de uma mensagem, enviada por um agente de grupo, definida anteriormente como recebimento de alerta de *timeout*. Essa mensagem é recebida apenas quando o usuário, representado pelo agente pessoal receptor, requisitou a formação de um grupo. Essa mensagem alerta ao agente pessoal que o tempo esperado pelo processo de formação do grupo de interesse coordenado por ele se esgotou. Assim, o agente pessoal deve encerrar sua comunicação com os agentes pessoais integrantes do grupo sugestivo recebido. Essa estratégia foi adotada para evitar que um agente pessoal fique esperando pela resposta de um outro agente pessoal que esteve ativado no ambiente, mas que foi desativado antes que pudesse responder às mensagens recebidas.

4.4
Instanciação do *f*Grupos

4.4.1
Questões de Implementação

Conforme ilustrado na figura 1, para que o *f*Grupos fosse reutilizado no desenvolvimento de aplicações de formação de grupos de interesse distintos, sete pontos de flexibilização foram definidos: (i) a base de dados utilizada; (ii) a heurística de definição de perfil do usuário; (iii) a heurística de definição de perfil do grupo; (iv) a estratégia de formação de grupos; (v) o algoritmo de *matching*; (vi) o algoritmo de parada; e, (vii) o formato de apresentação do grupo formado. Assim, a instanciação do *framework* se verifica através da implementação de todas as classes abstratas representantes destes *hot spots*.

A dificuldade de instanciar o *f*Grupos encontra-se não somente na complexidade da implementação das classes disponibilizadas como ponto de flexibilização, mas também no relacionamento estabelecido entre os seus métodos. Isto está relacionado com a existência de uma forte dependência entre as estruturas dos dados utilizados como parâmetro de entrada para a maioria desses métodos. Devido a isso, a implementação das classes abstratas que representam os pontos de flexibilização do *framework* deve ser feita simultaneamente e a

coerência que deve existir entre esses métodos é de responsabilidade do instanciador do *framework*. No entanto, algumas regras de implementação são estabelecidas para que a interface Web seja bem utilizada. Assim, recomenda-se que o *framework* seja instanciado seguindo duas etapas: a etapa de criação da base de dados utilizada, sendo esta relacional, semântica ou orientada a objetos; e, a etapa de implementação das classes abstratas que representam os pontos de flexibilização do *framework*.

A seguir são apresentadas as regras de implementação dos pontos de flexibilização do *framework f*Grupos.

4.4.1.1 Base de dados

O *hot spot* referente à base de dados utilizada é representado pela classe abstrata Facade do pacote dataBase, apresentado na figura 1.

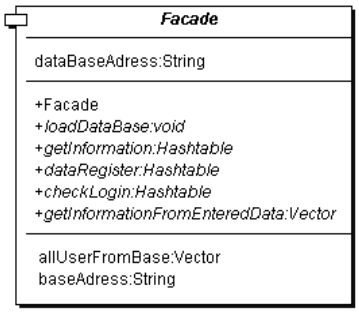

Figura 9: Classe abstrata Facade.

O primeiro passo é a criação da base de dados, seja semântica, relacional ou orientada a objetos. Posteriormente, os métodos abstratos da classe abstrata Facade devem ser implementados, conforme descrito a seguir.

a) Método loadDataBase

O método abstrato void loadDataBase() estabelece uma conexão entre o sistema e a base de dados, seja ela semântica, relacional ou orientada a objeto. Sua implementação é obrigatória para a instanciação do *framework*, pois o mesmo é utilizado pelo construtor da classe abstrata Facade, que é a primeira classe abstrata a ser instanciada no ato da configuração do *framework*. Por isso, o método void loadDataBase() deve ser o primeiro método abstrato a ser implementado.

Exemplo: A implementação desse método quando instanciado para a utilização de cada tipo de base de dados pode ser:

1. Base de Dados Relacional

```
Class.forName("com.mysql.jdbc.Driver");
```

2. Base de Dados Orientada a Objetos

```
this.prevayler = new SnapshotPrevayler(this, "/prevalenceBase");
```

3. Base de Dados Semântica

```
this.dataBaseAdress="file:///C:/projeto.daml";
this.model = new DAMLModelImpl();
this.model.read(this. dataBaseAdress);
```

b) Método getInformation

O método abstrato Hashtable getInformation (String query,Vector information) é utilizado para a obtenção de informações de uma base de dados relacional. Assim, somente pode ser implementado quando o *framework* for reutilizado para gerar uma instância que faça uso de uma base de dados relacional. Esse método abstrato é adicional ao *framework* (sua implementação não é obrigatória), e foi definido para atuar como suporte ao seu instanciador. Sua utilização pode ser necessária durante a implementação de alguns dos métodos abstratos indispensáveis para a instanciação do *framework*. Os parâmetros de entrada e o valor de retorno desse método devem seguir algumas regras de implementação necessárias para que a sua utilização seja possível na instanciação do *framework*, conforme descrito a seguir.

O parâmetro de entrada query é uma consulta SQL, que deve ser construída pelo instanciador do *framework,* utilizada para a obtenção de informações de uma base de dados relacional que segue os padrões SQL.

O parâmetro de entrada information é um vetor que contém os nomes das variáveis utilizadas na consulta SQL, que representam as informações a serem obtidas no banco de dados relacional.

O valor de retorno desse método é uma tabela *hash* composta pelas informações obtidas na base de dados, onde para cada entrada nesta tabela, a chave é o nome da variável que representa a informação e o valor é a informação

(uma *string* ou um vetor de *string* de tamanho um) ou as informações (vetor de *string* de tamanho n) associadas ao nome.

Exemplo: Este exemplo apresenta os parâmetros de entrada e o respectivo valor retornado deste método quando utilizado por uma instância que possui uma base de dados relacional. No exemplo a seguir o instanciador objetiva obter a identificação única do interesse Engenharia de Software e os usuários que possuem esse interesse. O valor retornado é uma tabela *hash* onde a primeira entrada armazena a identificação obtida referente ao interesse e, a segunda entrada armazena o CPF dos usuários que possuem o interesse fornecido na consulta.

Query: "SELECT idinteresse, usuario FROM atividade WHERE interesse=Engenharia de Software";

Information:

idinteresse	Usuario

Valor de Retorno:

idinteresse	[1]
usuario	[91518393187,11111111111]

c) Método getInformationFromEnteredData

O método abstrato Vector getInformationFromEnteredData (Vector enterData,Hashtable mainInformation) é utilizado para a obtenção de informações da base de dados (relacional, orientada a objetos ou semântica) a partir de um conjunto de dados de entrada. Sua implementação é obrigatória para a instanciação do *framework*, pois ele é utilizado pelo método execute da classe concreta DataRegister do pacote command para a obtenção dinâmica das informações necessárias para o cadastro do usuário e o cadastro do perfil do grupo. Isto é, a busca dessas informações na base de dados depende da seleção de dados anteriores fornecidos ao usuário através do formulário do seu cadastro ou do cadastro do perfil do grupo. Os parâmetros de entrada e o valor de retorno desse método devem seguir algumas regras de implementação necessárias para a sua utilização na instanciação do *framework*, conforme descrito a seguir.

O parâmetro de entrada enterData é um vetor de tabelas *hash*, as quais contêm, cada uma, informações referentes aos dados de entrada que foram selecionados pelo usuário, necessários para a obtenção das informações da base de dados. Essas tabelas *hash* foram obtidas do vetor de tabelas *hash* retornado pelo

método getInformationToUserRegister(). A elas foram acrescidos os dados cadastrais selecionados pelo usuário, os quais serão utilizados para a obtenção das informações na base de dados necessárias à finalização do cadastramento.

O parâmetro de entrada mainInformation é uma tabela *hash* composta pelas informações referentes ao dado que se deseja obter na base de dados. Esse tabela também foi obtida do vetor de tabelas *hash* retornado pelo método getInformationToUserRegister().

O valor de retorno desse método deve ser um vetor de *string* composto pelas informações obtidas na base de dados que serão utilizadas para compor um campo do formulário que deve ser selecionado para a finalização do processo de cadastro do usuário ou do perfil do grupo.

Exemplo: Este exemplo apresenta os parâmetros de entrada e o respectivo valor retornado deste método quando utilizado por uma instância que possui uma base de dados relacional. Nessa instância, um dos campos do formulário disponibilizado para o cadastro do usuário é o estado em que o mesmo reside. Assim, a seleção de um estado (SP, RJ) é necessária para a obtenção dinâmica das cidades que irão compor o campo cidade do formulário, que é o próximo campo a ser selecionado para o preenchimento do endereço. No exemplo a seguir, o parâmetro de entrada enterData é um vetor de tamanho um composto pela tabela *hash* que armazena informações referentes ao estado que foi selecionado (RJ) através do formulário fornecido ao usuário para o seu cadastro. As entradas dessa tabela e da tabela *hash* que representa o parâmetro de entrada mainInformation foram definidas durante a implementação do método getInformationToUserRegister(), descrito mais adiante. A utilização da maioria delas (entradas 1,2,5,6,7,8 e 9) é obrigatória para o funcionamento da interface Web. No entanto, as utilizações das entradas referentes às chaves tablename e primarykey não são obrigatórias e, foram definidas pelo instanciador do *framework* para armazenarem informações necessárias para a obtenção dos dados da base de dados relacional.

enterData:

Tabela *hash*

Tabela hash:

key	UF
UF	[RJ]
tablename	Estado
primarykey	idestado
needkey	new Vector()
empty	False
htmlType	ComboBox
validationType	new String()
informationDescription	Estado

mainInformation:

key	cidade
cidade	new Vector()
tablename	cidade
primarykey	idcidade
needkey	UF
empty	True
htmlType	ComboBox
validationType	New String()
informationDescription	Cidade

Valor de Retorno:

Rio de Janeiro	Petrópolis	Macaé

d) Método getAllUserFromBase

O método abstrato Vector getAllUserFromBase () é utilizado para a obtenção de todos os usuários cadastrados na base de dados. A chamada desse método é feita quando o *framework* é configurado como monousuário, e, neste caso, sua implementação torna-se obrigatória. Desta forma, as identificações de todos os usuários cadastrados na base de dados são obtidos, fazendo com que para cada um deles, um agente pessoal seja criado e iniciado. O valor de retorno desse método deve seguir algumas regras de implementação necessárias para a sua utilização na instanciação do *framework*, conforme descrito a seguir.

O valor de retorno desse método é um vetor composto pelas identificações pessoais dos usuários cadastrados na base de dados.

Exemplo: Este exemplo apresenta o valor retornado deste método quando utilizado por uma instância que identifica os usuários através do CPF. No exemplo a seguir, a base de dados possui apenas três usuários cadastrados com seus respectivos CPF.

Valor de Retorno:

91518393187	11111111111	11111111111

e) Método checkLogin

O método abstrato Hashtable checkLogin(String login) é utilizado para a verificação do *login* de entrada do usuário, representado pelo parâmetro de entrada login. Sua implementação é obrigatória para a instanciação do *framework*, pois o mesmo é utilizado pelo método execute da classe concreta VerifyLogin do pacote command.

A sua implementação exige que algumas regras de implementação sejam seguidas para que a interface Web seja bem utilizada. Tais regras estão associadas com os dados que compõem a tabela *hash*. Ou seja, caso a verificação seja realizada com sucesso, é obrigatório que a tabela *hash* retornada contenha uma chave denominada Login e o seu respectivo valor seja a identificação única do usuário em formato de *string*. Além disso, a tabela *hash* deve conter uma chave denominada Name com seu respectivo valor correspondente ao nome do usuário em formato de *string*. Se houver erro de verificação de *login* é obrigatório que a tabela *hash* contenha uma chave denominada Error e que o seu respectivo valor seja uma mensagem de erro em formato de *string*.

Exemplo: Este exemplo apresenta os parâmetros de entrada e o respectivo valor retornado deste método quando utilizado por uma instância que utiliza o CPF do usuário como *login* de entrada no ambiente e também como identificação única.

Login: 91518393187

Valor de Retorno:
Verificação realizada com sucesso:

Login	91518393187
Name	Angela Brigida Albarello

Verificação realizada sem sucesso:

Error	Usuário não cadastrado no ambiente

f) Método dataRegister

O método abstrato Hashtable dataRegister(Hashtable information) é utilizado para o registro dos dados cadastrais do usuário na base de dados após o preenchimento do formulário submetido ao mesmo. Sua implementação é obrigatória para a instanciação do *framework*, por ser utilizado pelo método

execute da classe concreta DataRegister do pacote command. Os parâmetros de entrada e o valor de retorno desse método devem seguir algumas regras de implementação necessárias para a sua utilização na instanciação do *framework*, conforme descrito a seguir.

O parâmetro de entrada information é uma tabela *hash* que contém os dados cadastrais do usuário. Esses dados cadastrais fornecidos pelo usuário são armazenados nessa tabela através da utilização de uma estrutura de dados pré-definida nas tabelas *hash* que compõem o vetor retornado pelo método Vector getInformationToUserRegister() (ver descrição desse método no próximo item). Assim, cada entrada da tabela information possui uma chave identificadora do dado cadastral referente a um campo do formulário preenchido e o seu respectivo valor. O nome da chave e o tipo de dado (*string* ou vetor) referente ao seu valor foram atribuídos seguindo o mesmo nome e tipo do dado definidos na segunda entrada de uma das tabelas *hash* pertencentes ao vetor retornado.

O valor de retorno é uma tabela *hash* definida através da utilização de algumas regras de implementação necessárias para que a interface Web seja utilizada. Tais regras estão associadas com os dados que a compõem. Ou seja, caso o registro dos dados cadastrais do usuário seja realizado com sucesso, é obrigatório que a tabela *hash* retornada contenha uma chave denominada Login e o seu respectivo valor seja a identificação única do usuário em formato de *string*. Se houver erro de verificação de *login* é obrigatório que a tabela *hash* contenha uma chave denominada Error e que o seu respectivo valor seja uma mensagem de erro em formato de *string*.

Exemplo:

Information:

nome	Ângela Brígida Albarello
CPF	91518393187
estado	[RJ]
cidade	[Rio de Janeiro]
interesse	[Engenharia de Software, Banco
competencia	[Java, MySQL,HTML,JSP]

Valor de Retorno:

Registro realizado com sucesso:

Login	91518393187

Registro realizado sem sucesso:

Error	Usuário já cadastrado no ambiente

4.4.1.2
Heurística de definição de perfil do usuário

O *hot spot* referente à heurística de definição de perfil do usuário utilizada é representado pela classe abstrata UserProfileDefineHeuristic do pacote model, apresentado na figura 10.

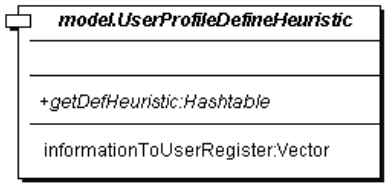

Figura 10: Classe abstrata UserProfileDefineHeuristic.

Os métodos abstratos da classe abstrata UserProfileDefineHeuristic devem ser implementados, conforme descrito a seguir.

a) Método getInformationToUserRegister

O método abstrato Vector getInformationToUserRegister() é utilizado para a obtenção das informações necessárias para o cadastro do usuário. Sua implementação é obrigatória para a instanciação do *framework*, porque é utilizado pelo método execute da classe concreta DataRegister do pacote command.

O retorno desse método é um vetor de tabelas *hash* definidas através da utilização de algumas regras de implementação necessárias para que a interface Web seja utilizada. Cada tabela *hash* armazena informações referentes a um campo do formulário que será fornecido ao usuário durante o seu cadastro. Essas informações são utilizadas para a construção do formulário, para a obtenção dinâmica dos dados cadastrais na base de dados e para o registro dos dados cadastrais fornecidos pelo usuário. Assim, a estrutura das tabelas *hash* que compõem o vetor retornado deve ser conforme descrito a seguir.

Estrutura da tabela *hash* de retorno:

Key	X - Nome da variável no formato de String
X	[x1,x2,...xn] ou new String() ou new Vector()
Needkey	[Y, Z] ou new Vector()
Empty	true ou false
htmlType	TextBox, ComboBox, CheckBox, Button ou TextArea
validationType	cep,CPF ou email
informationDescription	Descrição do campo do formulário

Figura 11: Tabela *hash* padrão composta pelas informações de cadastro.

Cada entrada na tabela *hash* acima é composta por uma chave identificadora e por um valor identificado pela chave, que são obrigatórios para o funcionamento da interface Web.

Key: O valor identificado pela chave composta pela *string* key, na primeira entrada da tabela, deve conter o nome da variável (X), armazenada em *string*, que representa a informação necessária para o cadastro do usuário. Esse mesmo nome é atribuído ao campo do formulário que contém a informação que deve ser selecionada ou que recebe a informação fornecida pelo usuário.

X: Na segunda entrada da tabela, a chave é composta pelo nome da variável (X), armazenada em uma *string*, que foi definida na primeira entrada da tabela. O valor identificado por essa chave pode ser composto por uma *string* vazia, por um vetor de *strings* vazio ou por um vetor de *strings* preenchidos com informações obtidas na base de dados. Esse valor é utilizado na construção do campo do formulário associado à variável X, o qual possui três situações distintas: (i) se o campo do formulário for utilizado para ser preenchido pelo usuário, então o valor da *string* deve ser vazio; (ii) se o campo do formulário for utilizado para ser selecionado pelo usuário a partir de informações obtidas na base de dados, então o vetor da *string* deve conter as informações associadas ao campo; e, (iii) se o campo do formulário for utilizado para ser selecionado pelo usuário a partir de informações que serão obtidas dinamicamente na base de dados após a seleção de um campo fornecido anterior, então o vetor da *string* deve ser vazio.

Needkey: O valor identificado pela chave composta pela string needkey, na terceira entrada da tabela, pode ser composto por: (i) um vetor de *strings* vazio, se as informações referentes ao campo do formulário não são obtidas dinamicamente; e, (ii) um vetor de *strings* compostos pelos nomes das variáveis atribuídas aos campos do formulário que necessitam de preenchimento, se for necessária a obtenção dinâmica das informações que compõem a variável contida nesta tabela *hash*.

Empty: O valor identificado pela chave composta pela *string* empty, na quarta entrada da tabela, pode ser composto pelas *strings*: (i) true, se o campo do formulário associado a variável contida nesta tabela for vazio, devendo ser preenchido pelo usuário; e, (ii) false, se o campo do formulário associado a

variável contida nesta tabela conter informações que devem ser selecionadas pelo usuário.

HTMLType: O valor identificado pela chave composta pela *string* htmlType, na quinta entrada da tabela, pode ser composto pelas *strings*: (i) TextBox, se o campo do formulário associado a variável contida nesta tabela for um *textBox*; (ii) ComboBox, se o campo do formulário associado a variável contida nesta tabela for um c*omboBox*; (iii) CheckBox, se o campo do formulário associado a variável contida nesta tabela for um c*heckBox*; (iv) Button, se o campo do formulário associado a variável contida nesta tabela for um b*utton*; e, (v) TextArea, se o campo do formulário associado a variável contida nesta tabela for um t*extArea*

ValidationType: O valor identificado pela chave composta pela string validationType, na sexta entrada da tabela, determina o tipo da validação de uma variável utilizada para a construção de um campo texto do formulário. O valor pode ser composto pelas *strings*: (i) Cep, se o campo texto do formulário for o cep. Então o código de validação *javascript* do *framework* verifica se o dado fornecido é um cep válido; (ii) CPF, se o campo texto do formulário for o CPF. Então o código de validação *javascript* do *framework* verifica se o dado fornecido é um CPF válido; e, (iii) Email, se o campo texto do formulário for o *email*. Então o código de validação *javascript* do *framework* verifica se o dado fornecido é um *email* válido.

Exemplo: O exemplo abaixo apresenta as tabelas *hash* que compõem o vetor retornado pelo método getInformationToUserRegister.

Primeira posição do vetor retornado:

key	nome
nome	new String()
needkey	new Vector()
empty	true
htmlType	TextBox
validationType	new String()
informationDescription	Nome

Segunda posição do vetor retornado:

key	CPF
CPF	new String()
needkey	new Vector()
empty	true
htmlType	TextBox
validationType	CPF
informationDescription	CPF

Terceira posição do vetor retornado:

key	UF
UF	[RJ,SP]
needkey	new Vector()
empty	false
htmlType	ComboBox
validationType	new String()
informationDescription	Estado

Quarta posição do vetor retornado:

key	cidade
cidade	new Vector()
needkey	UF
empty	true
htmlType	ComboBox
validationType	New String()
informationDescription	Cidade

b) Método getDefineHeuristic

O método abstrato Hashtable getDefHeuristic(String identification) é aplicado para a obtenção do perfil do usuário a partir da utilização de uma heurística de definição de perfil do usuário definida pelo instanciador do *framework*. Sua implementação é obrigatória para a instanciação do *framework*, por ser utilizado pelo método setup da classe concreta PersonalAgent do pacote model.

O parâmetro de entrada identification é uma *string* que determina a identificação única do usuário que está cadastrado no ambiente. Assim, os dados necessários para a definição do perfil do usuário podem ser obtidos na base de dados.

O valor de retorno desse método é uma tabela *hash* composta pelo perfil do usuário, o qual é definido pelo instanciador do *framework*. Devido a isso, a tabela *hash* pode armazenar qualquer tipo de informação, pois ela somente será utilizada pelos métodos abstratos do *framework*.

Exemplo: O exemplo abaixo apresenta uma forma de utilizar uma tabela *hash* para o armazenamento do perfil do usuário. O parâmetro de entrada desse método é o CPF do usuário. Seu valor de retorno é uma tabela *hash*. As informações armazenadas nessa tabela compõem o perfil do usuário e são de responsabilidade do instanciador do *framework*.

Parâmetro de entrada identification: 91518393187

Valor de Retorno:

Nome	Ângela Brígida Albarello
Cidade	Rio de Janeiro
UF	[RJ]
Interesse	[Engenharia de Software, Banco de Dados]
Competência	[Java, JSP, HTML, SQL]

4.4.1.3 Heurística de definição de perfil do grupo

O *hot spot* referente à heurística de definição de perfil do grupo utilizada é representado pela classe abstrata GroupProfileDefineHeuristic do pacote model, apresentado na figura a seguir.

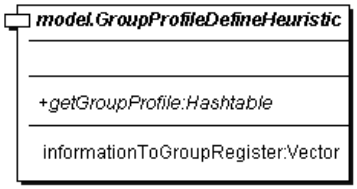

Figura 12: Classe abstrata GroupProfileDefineHeuristic.

Os métodos abstratos da classe abstrata GroupProfileDefineHeuristic devem ser implementados, conforme descrito a seguir.

a) Método getInformationToGroupRegister

O método abstrato Vector getInformationToGroupRegister() é utilizado para a obtenção das informações necessárias para o cadastro do perfil do grupo. Sua implementação é obrigatória para a instanciação do *framework*, pois ele é utilizado pelo método execute da classe concreta DataRegister do pacote command.

O retorno desse método é um vetor de tabelas *hash* definidas através da utilização de algumas regras de implementação necessárias para que a interface Web seja utilizada. Essas regras de implementação são as mesmas regras necessárias para a utilização do método getInformationToUserRegister(), conforme descritas anteriormente, acrescida da obrigatoriedade do fornecimento de um nome do grupo. Cada tabela *hash* armazena informações referentes a um campo do formulário que será fornecido ao usuário para o cadastro do perfil do grupo requerido por ele. Essas informações são utilizadas para a construção do

formulário, para a obtenção dinâmica dos dados cadastrais na base de dados e para a definição do perfil do grupo.

Exemplo: As tabelas abaixo apresentam os dados necessários para a construção do formulário de cadastro do perfil do grupo requisitado. O usuário deve fornecer o nome do grupo a ser formado e os interesses que compõem o perfil do grupo. Esses interesses são utilizados para a obtenção dos usuários que possuem interesse comum aos interesses que compõem o perfil do grupo.

Tabela h*ash* referente à primeira posição do vetor retornado:

key	nome
nome	new String()
needkey	new Vector()
empty	true
htmlType	TextBox
validationType	new String()
informationDescription	Nome do Grupo

Tabela *hash* referente à segunda posição do vetor retornado.

key	interesse
interesse	[Engenharia de Software, Banco de Dados]
needkey	new Vector()
empty	false
htmlType	checkBox
validationType	new String()
informationDescription	Interesse

Tabela *hash* referente à terceira posição do vetor retornado.

key	competencia
competencia	[Java, HTML, JSP, Oracle, MySQL]
needkey	interesse
empty	false
htmlType	checkBox
validationType	new String()
informationDescription	Competencia

b) Método getGroupProfile

O método abstrato Hashtable getGroupProfile(Hashtable information) é aplicado para a obtenção do perfil do grupo a partir da utilização de uma heurística de definição de perfil do grupo, definida pelo instanciador do *framework*. Sua implementação é obrigatória para a instanciação do *framework*,

por ser utilizado pelo método execute da classe concreta GroupFormation do pacote command.

O parâmetro de entrada information é uma tabela *hash* composta pelas informações fornecidas pelo usuário através do formulário de cadastro do perfil do grupo.

O valor de retorno desse método é uma tabela *hash* composta pelo perfil do grupo, o qual é definido pelo instanciador do *framework*. Devido a isso, a tabela *hash* pode armazenar qualquer tipo de informação, pois ela somente será utilizada pelos métodos abstratos do *framework*.

Exemplo: O exemplo abaixo apresenta o parâmetro de entrada information e a tabela hash que armazena o perfil do grupo, retornado pelo método getGroupProfile.

Information:

nome	grupo Java
interesses	[Engenharia de Software]
competências	[Java]

Valor de Retorno:

nome	grupo Java
key	interesses
interesses	[1]
key	competencias
competências	[1]

4.4.1.4 Estratégia de formação de grupos

O *hot spot* referente à estratégia de formação de grupos utilizada é representado pela classe abstrata GroupFormationEstrategy do pacote model, apresentado na figura a seguir.

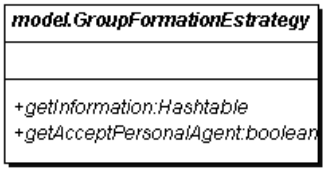

Figura 13: Classe abstrata GroupFormationEstrategy.

Existem várias estratégias que podem ser utilizadas para a formação de um grupo de interesse, as quais são de responsabilidade do instanciador do *framework*. Essa estratégia é composta pela escolha das informações trocadas

entre os agentes pessoais e pela heurística utilizada para a verificação de aceitação de um agente pessoal no grupo de interesse requisitado. Para que o *framework fG*rupos permitisse a interação entre os usuários como estratégia de formação de grupo para uma instância, o método concreto getInformationOnInterationWithUser foi definido para ser utilizado pelo método getInformation, descrito abaixo. Nessa instância, as informações pessoais do usuário avaliado a participar de um grupo não são obtidas a partir da base de dados. Ou seja, as informações são fornecidas pelo próprio usuário, que está ativo no ambiente, através do preenchimento de um formulário. Os dados que compõem o formulário são definidos pelo instanciador do *framework,* e devem seguir as mesmas regras de implementação utilizadas para o cadastro do usuário ou do perfil do grupo.

a) Método getInformation

O método abstrato Hashtable getInformation(AID aidAgP,AID aidAgG) é utilizado para a obtenção das informações pessoais dos usuários. Essas informações devem ser trocadas entre os agentes pessoais para que o processo de formação do grupo de interesse seja finalizado. Um agente pessoal faz a chamada deste método para obter as informações pessoais do usuário representado por ele, as quais foram requeridas por um outro agente pessoal que se comunica com este para a formação de um grupo de interesse. Sua implementação é obrigatória para a instanciação do *framework*, pois o mesmo é utilizado pelo método action da classe concreta PersonalAgentCommunication do pacote behaviour.

Existem dois tipos de obtenção de informações:

I) Obtenção estática

O instanciador do *framework* obtém as informações a partir da base de dados. Na instância gerada que utiliza esse tipo, os parâmetros de entrada aidAgP e aidAgG do método abstrato getInformation são utilizados, respectivamente, para a obtenção do perfil do usuário, através da chamada do método concreto Hashtable getPersonalAgentProfile(AID aidAgP), e para a obtenção do grupo, através da chamada do método concreto Group getGroup(AID agG). O valor de retorno é uma tabela *hash* que armazena as informações pessoais do usuário. Essas informações devem ser trocadas entre os agentes pessoais quando um deles as requisitam ao outro.

Exemplo: Um exemplo da tabela *hash* retornada pelo método getInformation quando utilizado para a obtenção estática das informações é apresentado a seguir.

Nesta instância, as informações referentes à localização do usuário e o tempo disponível são utilizados como estratégia de formação de grupo.

Valor de Retorno:

cidade	Rio de Janeiro
estado	RJ
tempo_disponivel	Segunda,Terça, Quarta, Domingo

II) Obtenção dinâmica

O instanciador do *framework* envia uma mensagem ao usuário para que este forneça as informações necessárias para a verificação de sua inclusão no grupo de interesse requisitado por outro usuário ativo no ambiente. Essa mensagem é composta por um formulário que deve ser preenchido e submetido ao remetente. Neste tipo de instância gerada, a interação entre os usuários é utilizada como estratégia de formação de grupo. Para isto, o método Hashtable getInformationOnInterationWithUser (Vector information, AID aidAgP) deve ser chamado no corpo de implementação do método abstrato getInformation.

O parâmetro de entrada information é um vetor composto por tabelas *hash* que armazenam informações necessárias para a construção do formulário que compõe a mensagem enviada ao usuário. Cada tabela *hash* deve ser definida seguindo as mesmas regras de implementação utilizadas para o cadastro do usuário e do perfil do grupo.

O parâmetro de entrada aidAgP, de ambos os métodos, refere-se ao ID do agente pessoal que faz a chamada do método abstrato getInformation. Sua utilização é necessária para o envio e obtenção do formulário de fornecimento de informação pessoal, que deve ser feita pelo usuário ativo no ambiente.

O retorno desse método é uma tabela *hash* composta pelas informações trocadas entre os agentes pessoais. Essa tabela *hash* é utilizada como valor de retorno para o método abstrato getInformation.

Exemplo: Este exemplo apresenta o vetor information utilizado como parâmetro de entrada para o método getInformationOnInterationWithUser e a tabela hash retornada pelo mesmo.

Information:	Tabela *hash*

Tabela hash:	Key	resposta
	resposta	[Sim,Não]
	needkey	new Vector()
	empty	False
	htmlType	Button
	validationType	new String()
	informationDescription	Deseja participar do grupo de interesse ?

Valor de Retorno:	resposta	Sim

b) Método getAcceptPersonalAgent

O método abstrato boolean getAcceptPersonalAgent (Hashtable information, Hashtable profile) é utilizado para verificar se o agente pessoal, que enviou as informações requeridas, pode ser inserido no grupo de interesse requisitado pelo agente pessoal que faz a chamada deste método. A sua implementação é obrigatória para a instanciação do *framework*, pois o mesmo é utilizado pelo método action da classe concreta PersonalAgentCommunication do pacote behaviour.

Existem várias formas de verificar a aceitação de um agente pessoal para a sua inserção no grupo de interesse em processo de formação. A forma tradicional é através da comparação entre o perfil do usuário representado pelo agente pessoal que faz a chamada desse método e as informações pessoais do agente avaliado. Uma segunda forma refere-se apenas a análise das informações pessoais recebidas por um agente pessoal. Desta forma, conclui-se que a implementação deste método é de responsabilidade do instanciador do *framework.*

O parâmetro de entrada information é uma tabela *hash* que contém as informações pessoais enviadas pelo agente pessoal que vai ser avaliado para suposta aceitação no grupo de interesse.

O parâmetro de entrada profile é uma tabela *hash* que contém o perfil do agente do grupo requisitado pelo agente pessoal que faz a chamada desse método.

O valor retornado é *true* se o agente for aceito para ser inserido no grupo de interesse, e *false* caso contrário.

4.4.1.5
Algoritmo de matching

O *hot spot* referente ao algoritmo de *matching* utilizado é representado pela classe abstrata GroupMatch do pacote model.

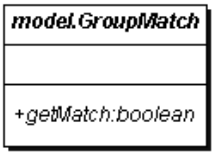

Figura 14: Classe abstrata GroupMatch.

O método abstrato getMatch da classe abstrata GroupMatch deve ser implementado conforme descrito a seguir.

a) Método getMatch

O método abstrato boolean getMatch(Hashtable personalProfile,Hashtable groupProfile) é utilizado pelo agente do grupo para efetuar o *matching* entre o perfil do agente pessoal e o perfil do grupo durante a obtenção do grupo sugestivo. A implementação deste método é obrigatória para a instanciação do *framework*, porque é utilizado pelo método action da classe concreta GroupAgent do pacote model.

O parâmetro de entrada personalProfile é uma tabela *hash* que contém o perfil do agente pessoal obtido para ser avaliado para a sua inserção no grupo sugestivo.

O parâmetro de entrada groupProfile é uma tabela *hash* que contém o perfil do agente do grupo o qual faz a chamada desse método durante o processo de formação do grupo sugestivo. O grupo sugestivo será enviado ao agente pessoal que requisitou o grupo de interesse.

O valor retornado deve ser *true* se o perfil do agente pessoal avaliado é adequado ao perfil do grupo. Desta forma, o agente pessoal é inserido no grupo sugestivo. Do contrário, o valor retornado deve ser *false*.

4.4.1.6
Algoritmo de parada

O *hot spot* referente ao algoritmo de parada utilizado é representado pela classe abstrata Stop do pacote model, apresentado na figura a seguir.

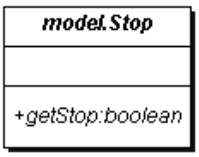

Figura 15: Classe abstrata Stop.

O método abstrato getStop da classe abstrata Stop deve ser implementado conforme descrito a seguir.

a) Método getStop

O método abstrato boolean getStop (Vector formedGroup, int nAgentsContacted) é utilizado para definir a parada do processo de formação do grupo sugestivo feito pelo agente do grupo. A sua implementação é obrigatória para a instanciação do *framework*, pois ele é utilizado pelo método setup da classe concreta GroupAgent do pacote model.

O parâmetro de entrada formedGroup é um vetor composto pelos agentes pessoais que foram inseridos no grupo sugestivo até o instante.

O parâmetro de entrada nAgentsContacted refere-se ao número de agentes que foram obtidos da classe CommunityManager e verificados para a inserção no grupo sugestivo.

O valor retornado deve ser true se o grupo formado já é o suficiente. Caso contrário, o valor retornado deve ser false.

4.4.1.7
Formato de apresentação do grupo formado

O *hot spot* referente ao formato de apresentação do grupo formado utilizado é representado pela classe abstrata Exit do pacote model. Os métodos abstratos da classe abstrata Exit devem ser implementados, conforme descrito a seguir.

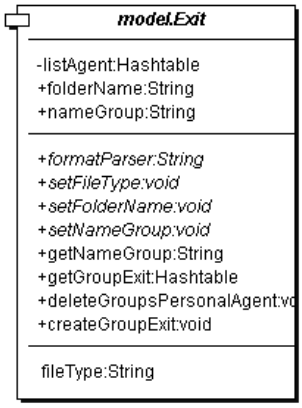

Figura 16: Classe abstrata Exit.

a) Método formatParser

O método abstrato String formatParser(AID aidAgP,Group gr) é utilizado para gerar a saída do grupo formado em um formato de dados (xml, html, etc.) definido na instanciação do *framework*. A sua implementação é obrigatória para a instanciação do *framework*, porque ele é utilizado pelo método concreto createGroupExit da classe abstrata Exit do pacote model, o qual é utilizado por um agente do grupo.

O parâmetro de entrada aidAgP refere-se ao ID do agente pessoal que requisitou o grupo formado. Através do ID do agente pessoal, o perfil do usuário pode ser obtido da classe CommunityManager através da chamada do método concreto Hashtable getPersonalAgentProfile(AID aidAgP).

O parâmetro de entrada gr refere-se ao grupo de interesse formado. Através dele o perfil do grupo e os integrantes do grupo podem ser obtidos.

O valor retornado é uma *string* composta pelas informações referentes ao grupo de interesse, formatadas de acordo com o formato utilizado. Em seguida essas informações são armazenadas em um arquivo em disco, o qual possui o nome do grupo fornecido pelo usuário que requisitou a sua formação.

b) Método setNameGroup

O método abstrato void setNameGroup(Group gr) é utilizado para definir o nome do arquivo de saída que contém as informações formatadas referentes ao grupo formado. O nome do arquivo é o mesmo nome do grupo que foi fornecido pelo usuário que cadastrou o perfil do grupo. A implementação desse método é obrigatória para a instanciação do *framework*, pois ele é utilizado pelo método concreto createGroupExit da classe abstrata Exit do pacote model.

c) Método setFileType

O método abstrato void setFileType() é utilizado para definir a extensão (txt, html, xml, etc.) do arquivo de saída que contém as informações formatadas referentes ao grupo formado. A sua implementação é obrigatória para a instanciação do *framework*, porque também é utilizado pelo método concreto createGroupExit da classe abstrata Exit do pacote model, o qual é utilizado por um agente do grupo.

d) Método setFolderName

O método abstrato void setFolderName() é utilizado para definir o nome da pasta a ser criada para o armazenamento do arquivo de saída que contém as informações formatadas referentes ao grupo formado. A sua implementação é obrigatória para a instanciação do *framework*, porque também é utilizado pelo método concreto createGroupExit da classe abstrata Exit do pacote model.

4.4.2 Questões de Configuração

A configuração necessária para a instanciação do *framework* deve ser feita no corpo do método void init(ServletConfig config) da classe concreta Control do pacote control.

Inicialmente, o instanciador do *framework* deve definir se o sistema instanciado será monousuário ou multiusuário. Para isto, o método void setInitPersonalAgentFromBaseCommand(String command) deve ser configurado através de um parâmetro de entrada. Quando o sistema for monousuário, o parâmetro command recebe o valor true. Caso contrário, o parâmetro command recebe o valor false.

Por último, o instanciador deve implementar as classes abstratas e configurar os métodos de inicialização destas classes concretas criadas. Cada método de inicialização de uma classe concreta criada deve ter como parâmetro de entrada um objeto único instanciado. Assim, os métodos setFacade (new Object()), setUserProfileDefineHeuristic (new Object()), setGroupProfileDefineHeuristic (new Object()), setExit(), setStop(), setGroupMatch() e setGroupFormationEstrategy() devem ser configurados através dos seus respectivos objetos de entrada.