UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE CIÊNCIAS E TECNOLOGIA
CURSO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Um Modelo para o Desenvolvimento de Aplicações Baseadas em Agentes Móveis

Fabiana Paulino Guedes

Campina Grande – PB
Agosto – 2002

UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE CIÊNCIAS E TECNOLOGIA
CURSO DE PÓS GRADUAÇÃO EM INFORMÁTICA

Um Modelo para o Desenvolvimento de Aplicações Baseadas em Agentes Móveis

Fabiana Paulino Guedes

Dissertação submetida à Coordenação de Pós-Graduação em Informática do Centro de Ciências e Tecnologia da Universidade Federal de Campina Grande como requisito parcial para a obtenção do grau de Mestre em Ciências (MSc).

Área de Concentração: Ciências da Computação
Linha de Pesquisa: Engenharia de Software

Orientadora: Patrícia Duarte de Lima Machado

Campina Grande – PB
Agosto - 2002

Agradecimentos

A Deus e a minha família, pela oportunidade e condições de realizar este trabalho.

A minha orientadora, Patrícia, pela dedicação, confiança, apoio e incentivo.

A meu noivo, André, por seu amor, compreensão e força.

As minhas amigas, Beti, Lady e Raquel, por sempre estarem ao meu lado com palavras reconfortantes e animadoras.

A Vivianne, pelos conhecimentos compartilhados e auxílio sempre constante.

A Emerson, pela amizade, apoio e disponibilidade em ajudar.

Aos professores, aos colegas do DSC e a todos aqueles que colaboraram para a realização deste trabalho.

Sumário

1 Introdução _ 1

1.1 Objetivos _ 2

1.2 Estrutura da Dissertação _ 3

2 Introdução a Agentes Móveis _ 4

2.1 Surgimento do Paradigma de Agentes Móveis _ 4

2.2 Benefícios _ 5

2.3 Áreas de Aplicação _ 6

2.4 Problemas _ 7

2.5 Plataformas de Agentes Móveis _ 8

2.6 Padrão para a interoperabilidade de plataformas _ 9

2.7 Conceitos Básicos _ 10

2.7.1 Ambiente de Agentes Distribuído _ 10

2.7.2 Agente _ 11

2.7.3 Agente Móvel _ 11

2.7.4 Agente Estacionário _ 11

2.7.5 Agentes *versus* Objetos _ 11

2.7.6 Ciclo de Vida do Agente _ 12

2.7.6.1 Criação _ 12

2.7.6.2 Comunicação _ 12

2.7.6.3 Migração _ 13

2.7.6.4 Clonagem _ 14

2.7.6.5 Persistência _ 14

2.7.6.6 Destruição _ 15

2.7.7 Estados do Agente _ 15

2.7.8 Agência _ 15

2.7.8.1 Núcleo da Agência _ 16

2.7.8.2 Lugar _ 17

2.7.9 Região _ 17

3 Modelo para o Desenvolvimento de Aplicações Baseadas em Agentes Móveis _ 18

3.1 Motivações e Trabalhos Correlatos ______ **18**

3.2 O Modelo e seu Ciclo de Vida ______ **20**

3.3 Estudo de Caso ______ **22**

3.4 Considerações Finais ______ **23**

4 Fase de Análise ______ ***24***

4.1 Objetivo ______ **24**

4.2 Entradas ______ **24**

4.3 Tarefas ______ **25**

4.4 Artefatos ______ **25**

4.4.1 Casos de Uso Expandidos ______ 25

4.4.2 Modelo Conceitual ______ 30

4.4.3 Glossário ______ 32

4.4.4 Diagramas de Seqüência do Sistema ______ 34

4.4.5 Diagrama de Atividades ______ 35

4.4.6 Contratos ______ 37

4.5 Validação ______ **48**

4.6 Considerações Finais ______ **48**

5 Projeto Arquitetural ______ ***50***

5.1 Objetivo ______ **50**

5.2 Entradas ______ **50**

5.3 Tarefas ______ **50**

5.4 Artefatos ______ **51**

5.4.1 Projeto Arquitetural ______ 51

5.4.1.1 Descrição da Arquitetura do Sistema ______ 52

5.4.1.2 Descrição do comportamento dos agentes ______ 54

5.4.1.3 Plataforma de Agentes Móveis Escolhida ______ 57

5.5 Validação ______ **57**

5.6 Considerações Finais ______ **58**

6 Projeto Detalhado Independente de Plataforma ______ ***59***

6.1 Objetivo ______ **59**

6.2 Entradas ______ **59**

6.3 Tarefas ______ **59**

6.4 Artefatos ______ **60**

6.4.1 Diagramas de Interação ______ 60

6.4.1.1 Diagramas de interação da operação de sistema *gerarFormRevisao* _ 63

6.4.1.2 Diagramas de interação da operação de sistema *redirecionarAgenteFormRevisao* ______ 67

6.4.1.3 Diagramas de interação da operação de sistema *finalizarRevisao* ___ 68

6.4.1.4 Diagramas de interação da operação de sistema *aprovarRevisao* ___ 69

6.4.2 Diagrama de Classes ______ 70

6.4.3 Esquema de Banco de Dados ______ 75

6.5 Validação ______ **76**

6.6 Considerações finais ______ **76**

7 Projeto Detalhado Dependente de Plataforma ______ ***78***

7.1 Objetivo ______ **78**

7.2 Entradas ______ **78**

7.3 Tarefas ______ **78**

7.4 Artefatos ______ **79**

7.4.1 Diagramas de Interação Dependentes de Plataforma ______ 79

7.4.2 Simulação da Migração Forte ______ 83

7.4.3 Diagrama de Classes ______ 84

7.5 Validação ______ **86**

7.6 Considerações Finais ______ **86**

8 Conclusões ______ ***87***

8.1 Contribuições ______ **89**

8.2 Trabalhos futuros ______ **90**

9 Apêndice A - Implementação ______ ***91***

9.1 Classe AplicacaoConferencia ______ **91**

9.2 Interface *ServidorConferenciaIF* ______ **94**

9.3 Classe *Conferencia* ______ **96**

9.4 Interface *AgenteCoordenadorIF* 103

9.5 Interface *AgenteMestreIF* 104

9.6 Classe *AgenteCoordenador* 105

9.7 Interface *AgenteFormRevisaoIF* 110

9.8 Classe *AgenteFormRevisao* 112

9.9 Interface ItinerarioIF 128

9.10 Classe Itinerario 129

10 Referências bibliográficas 132

Lista de Figuras

Figura 2.1: Ambiente Distribuído 10

Figura 2.2: Ciclo de vida de um agente móvel 12

Figura 2.3: Simulação da migração forte 14

Figura 2.4: Diagrama de estados do agente 15

Figura 3.1: Modelo de Desenvolvimento Baseado em Agentes Móveis 20

Figura 4.1: Diagrama de caso de uso parcial para o Sistema de Apoio a Comitês de Programa em Conferências 26

Figura 4.2: Modelo conceitual parcial do estudo de caso 31

Figura 4.3: Modelo conceitual do AgenteFormRevisao 32

Figura 4.4: Diagrama de seqüência do caso de uso Revisar artigo 34

Figura 4.5: Diagrama de atividades do caso de uso "Iniciar processo de revisão de um artigo". 35

Figura 4.6: Diagrama de atividades do caso de uso "Redirecionar Formulário de Revisão". 35

Figura 4.7: Diagrama de atividades do caso de uso "Revisar artigo" 36

Figura 4.8: Diagrama de atividades do caso de uso "Aprovar revisão" 36

Figura 4.9: Operações do sistema 38

Figura 5.1: Decomposição em camadas do sistema 52

Figura 5.2: Configuração física do sistema 53

Figura 5.3: Padrão itinerário 54

Figura 5.4: Padrão movimento estrela 55

Figura 5.5: Padrão ramificação 55

Figura 5.6: Diagrama de comportamento 56

Figura 6.1: Representação gráfica de uma região em um diagrama de interação 60

Figura 6.2: Representação gráfica de uma agência em um diagrama de interação 61

Figura 6.3: Representação gráfica de um agente móvel 61

Figura 6.4: Representação gráfica de uma migração 61

Figura 6.5: Representação gráfica de uma clonagem 62

Figura 6.6: Representação gráfica de um agente estacionário 62

Figura 6.7: Diagrama de sequência da operação de sistema gerarFormRevisao – Parte 1 64

Figura 6.8: Diagrama de colaboração da operação de sistema gerarFormRevisao – Parte 1 64

Figura 6.9: Diagrama de sequência da operação de sistema gerarFormRevisao – Parte 2 65

Figura 6.10: Diagrama de colaboração da operação de sistema gerarFormRevisao – Parte 2 66

Figura 6.11: Diagrama de sequência da operação de sistema gerarFormRevisao – Parte 3 67

Figura 6.12: Diagrama de colaboração da operação de sistema gerarFormRevisao – Parte 3 67

Figura 6.13: Diagrama de sequência da operação de sistema redirecionarAgenteFormRevisao 68

Figura 6.14: Diagrama de sequência da operação de sistema finalizarRevisao 69

Figura 6.15: Diagrama de sequência da operação de sistema aprovarRevisao – Parte 1 69

Figura 6.16: Diagrama de sequência da operação de sistema aprovarRevisao – Parte 2 70

Figura 6.17: Diagrama de classes de objetos ______ *71*

Figura 6.18: Representação Gráfica da Classe de um Agente Móvel em um Diagrama de Classes ______ *71*

Figura 6.19: Representação Gráfica da Classe de uma Agência em um Diagrama de Classes ______ *72*

Figura 6.20: Representação Gráfica da Classe de um Agente Estacionário em um diagrama de Classes ______ *72*

Figura 6.21: Diagrama de classes de agentes ______ *72*

Figura 6.22: Diagrama de classes AgenteFormRevisao ______ *74*

Figura 6.23: Diagrama de classes Mestre-Escravo Intinerante ______ *75*

Figura 7.1: Diagrama de sequência dependente de plataforma da operação gerarFormRevisao – Parte 1 ______ *80*

Figura 7.2: Diagrama de sequência dependente de plataforma da operação gerarFormRevisao – Parte 2 ______ *81*

Figura 7.3: Diagrama de sequência dependente de plataforma da operação gerarFormRevisao – Parte 3 ______ *81*

Figura 7.4: Diagrama de sequência dependente de plataforma da operação redirecionarAgenteFormRevisao *82*

Figura 7.5: Diagrama de sequência dependente de plataforma da operação finalizarRevisao ______ *82*

Figura 7.6: Diagrama de sequência dependente de plataforma da operação aprovarRevisao – Parte 2 ______ *82*

Figura 7.7: Diagrama de atividades do método `executarTarefa( )` da classe AgenteFormRevisao ______ *83*

Figura 7.8 : Diagrama de classes dependente de plataforma ______ *84*

Figura 7.9 : Diagrama de classes dependente de plataforma do AgenteCoordenador ______ *85*

Figura 7. 10: Diagrama de classes dependente de plataforma do AgenteFormRevisao ______ *85*

Figura 9.1: G.U.I. do Agente Coordenador ______ *105*

Figura 9.2: G.U.I. do AgenteFormRevisao em distribuição ______ *114*

Figura 9.3: G.U.I. do AgenteFormRevisao em revisão. ______ *115*

Figura 9.4: G.U.I. do AgenteFormRevisao em aprovação. ______ *116*

Resumo

Recentemente, um novo paradigma para a construção de aplicações distribuídas em larga escala tem emergido: agentes móveis. Um agente móvel é uma entidade de software autônoma que é capaz de migrar entre localizações físicas da rede e continuar a sua execução do ponto em que parou antes da sua migração. Até o momento, a maioria das aplicações baseadas em agentes móveis tem sido criada de forma *ad-hoc*, seguindo pouca ou nenhuma metodologia. Isto se deve ao fato de que os modelos de processos atuais não são suficientes para cobrir todos os aspectos de mobilidade na modelagem, projeto e verificação de tais aplicações. Neste trabalho apresentamos um modelo para o desenvolvimento de aplicações baseadas em agentes móveis que combina o processo iterativo e incremental com o uso de padrões de projeto de agentes móveis e aspectos a serem considerados nas atividades das fases de análise e projeto. Artefatos são produzidos usando uma extensão de UML (*Unified Modeling Language*) que inclui aspectos relativos à mobilidade para a modelagem do sistema. A fim de ilustrar a aplicabilidade do modelo um estudo de caso é apresentado.

Abstract

More recently, a new paradigm for developing large-scale distributed applications has emerged: mobile agents. A mobile agent is an autonomous software entity that can migrate to different physical locations and continue its execution at the point where it stopped before migration. Up to now, the majority of existing mobile agent-based applications have been created in an *ad-hoc* way, following little or no methodology. One reason is that current process models do not properly cover requirements and aspects of mobility in the modeling, designing and verification of such applications. We present a model for developing mobile agent-based applications that combines the standard iterative and incremental unified process with the use of mobile agent design patterns and issues that should be considered in the activities of analysis and design. Artifacts are produced using an extension of the Unified Modelling Language (UML) that copes with mobility. To illustrate the applicability of the model a case study is presented.

1 INTRODUÇÃO

A mobilidade de agentes tem sido apontada como um conceito importante e proeminente para o desenvolvimento de aplicações distribuídas para redes em área de longo alcance e em redes contendo dispositivos móveis tal como a Internet. Em particular, o crescente interesse na tecnologia de agentes tem sido motivado pelo seu grande potencial de aplicação em diversas áreas, incluindo comércio eletrônico, busca e recuperação de informações, serviços de redes de telecomunicações, assistentes pessoais, assistentes gerenciais e administração de sistemas [MDW99, HB99, Bra00, LO98, BHM98]. Além do mais, tecnologias, padrões e metodologias convencionalmente usados para o desenvolvimento de aplicações distribuídas são limitados ao fornecerem o grau de configurabilidade, escalabilidade e customização necessários a tais tipos de aplicações [FPV98].

Agentes são unidades de software que executam tarefas em nome de uma pessoa ou de uma organização e que possuem a capacidade de interagir com o seu ambiente de execução de forma autônoma e assíncrona. Um agente pode possuir outras características alternativas, tais como inteligência e/ou mobilidade. Entretanto, este trabalho de dissertação trata apenas de agentes móveis - entidades de software autônomas capazes de migrar entre localizações físicas da rede e continuarem a sua execução do ponto em que haviam parado antes da sua migração. No desenvolvimento de aplicações distribuídas, agentes móveis prometem diversas vantagens tais como redução do tráfego da rede, acesso local aos recursos, execução assíncrona e autônoma, robustez e tolerância a falhas [LO98].

Um número de estudos de caso tem demonstrado a aplicabilidade de agentes móveis [VBML99, HB99, VZM00, BM00, KRSW01, GOP02]. No entanto, até agora, a maioria das aplicações baseadas em agentes móveis existentes foram criadas de forma *ad-hoc*, seguindo pouca ou nenhuma metodologia [KRSW01]. Isto se deve ao fato de que os métodos e técnicas O.O. (Orientadas a Objetos) existentes para o desenvolvimento de aplicações baseadas em componentes distribuídos não são suficientes para cobrir todos os aspectos de mobilidade na modelagem, projeto e verificação de sistemas baseados em agentes móveis.

Algumas abordagens em direção a uma metodologia para o desenvolvimento de sistemas multi-agentes [WJK99, OPB00, MKC01, GSLM01] já foram propostas, abordando, no entanto, apenas padrões de interação e cooperação. Embora alguns esforços relativos à mobilidade já tenham sido realizados, estes tratam apenas de atividades específicas, técnicas e notações [Gro97, YBF98, Car99, TOH99, KRSW01, CLZ01], tornando-se fundamental a

existência de uma abordagem que a considere adequadamente em um processo de desenvolvimento como um todo.

1.1 Objetivos

O principal objetivo deste trabalho é fornecer um modelo para o desenvolvimento de aplicações baseadas em agentes móveis de acordo com a disciplina de engenharia de software.

O modelo a ser apresentado não tem a pretensão de sugerir um processo totalmente inovador, mas sim adicionar considerações sobre aspectos relevantes à construção de aplicações baseadas em agentes móveis especificamente nas fases de análise e projeto. Este modelo consiste na adaptação do processo de desenvolvimento de software apresentado por Larman [Lar02], que é uma versão do processo unificado; utiliza também a aplicação de padrões de projeto de agentes móveis como sugerido por Tahara et al [TOH99] e de uma extensão de UML que lida com mobilidade proposta por Klein et al [KRSW01].

O diferencial do modelo proposto encontra-se na fase de projeto detalhado que é dividido em duas fases, uma independente e a outra dependente de plataforma. Isto se deve ao fato de que aplicações baseadas em agentes móveis geralmente são construídas em diferentes plataformas, que por sua vez estão evoluindo rapidamente. A primeira fase trata de todo o projeto lógico da aplicação, sem entrar em detalhes relacionados a uma plataforma de agentes móveis específica. Enquanto que a segunda apenas refina a primeira adicionando os detalhes de projeto relacionados a uma plataforma específica, a escolhida para implementar a aplicação. Desta forma, se durante ou após a execução do projeto ocorrer uma mudança na escolha da plataforma de agentes móveis, apenas a segunda fase do projeto detalhado precisará ser refeita.

A fim de refinar o modelo proposto, bem como analisar a viabilidade do mesmo, realizamos um estudo de caso através do desenvolvimento de uma aplicação para a Internet sugerida por [Car99]: Sistema de Apoio às Atividades de Comitês de Programa em Conferências. Esta aplicação gerencia as atividades de um comitê de programa de uma conferência, tais como a submissão de artigos, o processo de revisão e a notificação para os autores sobre a aceitação ou rejeição dos artigos. Esta aplicação foi escolhida pelo fato de permitir a exploração de diversos aspectos, tais como a mobilidade e a possibilidade de execução desconectada que se tornam possíveis com o uso de agentes móveis.

1.2 Estrutura da Dissertação

Esta dissertação está dividida em oito capítulos, sendo o primeiro deles a presente introdução. O Capítulo 2 introduz os principais conceitos do paradigma de agentes móveis. No Capítulo 3, o modelo para o desenvolvimento de aplicações baseadas em agentes móveis é proposto. Os Capítulos 4, 5,6 e 7 apresentam as fases do modelo abordadas neste trabalho. Por fim, as conclusões e sugestões para trabalhos futuros serão apresentadas no Capítulo 8.

2 INTRODUÇÃO A AGENTES MÓVEIS

Este capítulo tem o objetivo de familiarizar o leitor com os principais conceitos do paradigma de agentes móveis. A Seção 2 aborda o surgimento do paradigma de agentes móveis considerando-o uma conseqüência da evolução dos paradigmas de desenvolvimento de aplicações distribuídas tradicionais. As Seções 3, 4 e 5 apresentam respectivamente os benefícios, áreas potenciais de aplicação e problemas do paradigma de agentes móveis. Já a Seção 6, cita as principais plataformas de agentes móveis baseadas em Java e a Seção 7 apresenta o primeiro padrão de interoperabilidade de sistemas de agentes móveis. Por fim, a Seção 8 define os principais conceitos do paradigma de agentes móveis.

2.1 Surgimento do Paradigma de Agentes Móveis

O paradigma de agentes móveis surge como uma conseqüência da evolução dos paradigmas de desenvolvimento de aplicações distribuídas tradicionais (REV) [VBB00], que já não oferecem o grau de configuração, extensibilidade e customização necessários às aplicações distribuídas em larga escala [FPV98].

O paradigma de aplicações distribuídas mais adotado é o cliente servidor. Nele as aplicações são estruturadas como um conjunto de processos cliente e servidor que interagem através da troca de mensagens (síncronas ou assíncronas) ou através de chamadas de procedimentos remotos (RPC). O modelo de comunicação RPC permite que o processo cliente chame um procedimento em uma máquina remota, ficando a cargo da implementação de RPC na máquina cliente capturar a chamada ao procedimento remoto, encapsular seus parâmetros em mensagens de rede e então enviar estas mensagens para a máquina remota. A implementação de RPC na máquina remota recebe estas mensagens, recupera os parâmetros e então chama localmente o procedimento. Uma vez terminada a execução do mesmo, acontece o processo inverso para enviar o resultado até a máquina cliente.

Um outro modelo de comunicação proposto para sistemas distribuídos é o de Avaliação Remota (REV). Neste modelo, ao invés do cliente chamar um procedimento remoto, ele envia seu próprio código de procedimento para o servidor solicitando sua avaliação remota e o retorno do resultado. REV é mais flexível que RPC, pois o conjunto de serviços que podem ser invocados remotamente não precisa ser previamente definido. Uma outra vantagem apresentada por este modelo é que, para algumas aplicações, REV pode reduzir o montante de comunicação necessário à realização de uma tarefa. Esta redução é obtida graças à

possibilidade de se enviar o programa que manipula um conjunto de dados até a máquina onde os mesmos estão localizados, enquanto que em RPC, por exemplo, os dados é que devem ser obrigatoriamente movidos para a máquina onde se encontra o programa.

O paradigma de agentes móveis apresenta características dos dois modelos citados acima. Neste paradigma, um agente móvel migra de uma máquina a outra da rede carregando consigo os seus dados, como em RPC, e o seu código, como em REV. No entanto, ao contrário destes dois modelos, um agente é autônomo, não precisando retornar imediatamente ao nó de origem após a sua execução no primeiro nó da rede visitado. Ele pode, sempre com o objetivo de executar a sua tarefa, migrar por outros nós da rede antes de retornar ao nó de origem.

2.2 Benefícios

A adoção da tecnologia de agentes móveis para o desenvolvimento de aplicações distribuídas está sendo motivada pelos seguintes benefícios [LO98]:

- **Redução do tráfego da rede.** Diferentemente do modelo cliente servidor, que requer inúmeras interações para executar uma tarefa (o que é especialmente verdade quando medidas de segurança estão sendo utilizadas), o modelo de agentes móveis permite empacotar toda a comunicação e enviá-la para a máquina remota, onde as interações ocorrerão localmente. Agentes móveis também são úteis para reduzir o fluxo de dados na rede. Por exemplo, quando grandes volumes de dados estão armazenados em máquinas remotas, pode-se enviar um agente móvel até o mesmo, ao invés de se transferir todos os dados para a máquina de origem.
- **Eliminação da latência da rede.** Sistemas críticos necessitam de respostas em tempo real para mudanças no ambiente. O controle desses sistemas através de uma rede substancialmente grande ocasiona uma latência inaceitável. Agentes móveis oferecem uma solução, pois podem ser despachados pelo controlador central para realizarem suas tarefas localmente.
- **Execução assíncrona e autônoma.** Uma vez disparados, agentes móveis tornam-se independentes da máquina de origem, operando de forma assíncrona e autônoma. Esta vantagem é particularmente importante no caso de dispositivos computacionais móveis, como *notebooks* e assistentes pessoais onde não é técnica e economicamente viável a suposição de conectividade contínua.

- **Adaptação dinâmica.** Agentes móveis possuem a habilidade de perceber mudanças no ambiente de execução e reagir autonomamente. Por exemplo, um agente móvel, executando em um assistente pessoal, pode migrar para uma estação da rede fixa ao detectar que a bateria do assistente não dispõe de energia suficiente para o término da execução de sua tarefa.
- **Execução em diversas arquiteturas.** Redes de computadores, geralmente são heterogêneas, tanto na perspectiva de *hardware* como na de *software*. Agentes móveis são independentes da máquina e também da rede, sendo dependentes apenas do seu ambiente de execução, não dificultando a integração de sistemas.
- **Robustez e tolerância a falhas.** A habilidade dos agentes móveis de reagirem dinamicamente a situações e eventos desfavoráveis facilita a construção de sistemas distribuídos robustos e tolerantes a falhas. Se uma máquina está para ser desligada, todos os agentes em execução na máquina podem ser advertidos para que eles possam se despachar e continuar suas tarefas em outra máquina da rede.

2.3 Áreas de Aplicação

Diversas são as áreas que podem se beneficiar com o uso da tecnologia de agentes móveis, entre elas podemos citar:

- **Comércio Eletrônico.** Freqüentemente mencionada como uma área de uso potencial de agentes móveis. A infra-estrutura do comércio eletrônico está emergindo rapidamente. Agentes móveis podem localizar serviços ou produtos, comparar ofertas e negociar com os fornecedores em nome dos seus criadores [BHM98; MDW99; FPV98];
- **Busca e recuperação de informação.** Esta atividade pode ser realizada por agentes de pesquisa móveis na Internet. Exemplo de uma arquitetura básica para este tipo de aplicação: cada nó da rede contém um servidor de informações que pode executar o agente e fazer o processamento da busca. Os agentes coletam a informação em cada nó e retornam ao usuário com os resultados processados da busca. O paralelismo permite ainda que vários clones sejam disparados verificando concorrentemente a atualização de *sites*. Entretanto, a utilização de agentes móveis só é interessante se a plataforma de agentes estiver instalada nos servidores, do contrário o agente faz a busca remota, como em qualquer mecanismo de busca comum [Bra00; BHM98; MDW99];

- **Serviços de redes de telecomunicações.** Suporte e gerência de serviços de telecomunicações avançados são caracterizados pela customização do usuário e pela configuração dinâmica da rede. Este é um dos campos potenciais para o uso da tecnologia de agentes móveis, visto que ela possui um grau de configuração e customização superior ao das tecnologias tradicionais [BHM98, FPV98, HB99, LO98];
- **Assistentes pessoais.** Este foco de aplicação se destina a auxiliar o ser humano nas várias tarefas pessoais e profissionais. Nestes tipos de aplicação um agente móvel realiza tarefas na rede em nome do seu criador. Por exemplo, se um usuário deseja marcar uma reunião com um grupo de pessoas, ele poderia mandar um agente móvel para interagir com os agentes representantes de cada membro do grupo. E os agentes, em nome dos seus representantes, é que iriam negociar e estabelecer um horário para a reunião [LO98];
- **Assistentes gerenciais.** Agentes móveis podem ser responsáveis por gerenciar processos de negócio e coordenar atividades em grupo (*Workflow*) aumentando a eficiência do trabalho através da troca automática de informações mesmo quando os membros do grupo estão desconectados ou não disponíveis [Bra00].
- **Administração de Sistemas.** Agentes móveis podem ser utilizados para automatizar tarefas de rotina e verificar sistemas e redes periodicamente realizando uma avaliação inicial dos problemas localmente, antes da intervenção de um operador humano [MDW99].

2.4 Problemas

Por ainda ser recente, a tecnologia de agentes móveis ainda apresenta muitas limitações e desafios que precisam ser superados:

- **Segurança.** Em geral, mas em particular no âmbito de aplicações baseadas em agentes móveis colocam-se questões importantes do ponto de vista da segurança. O fato de um agente poder mover-se para uma máquina remota, e aí se executar, exige mecanismos de segurança de diferentes níveis, como segurança do local remoto com relação ao agente móvel; do agente com relação ao local remoto; e segurança no próprio canal de comunicação. A questão de segurança em agentes móveis ainda possui muitos problemas não solucionados e é uma área aberta de pesquisa [MDW99; Sil99].

- **Padrões.** Sem padrões comuns de uma tecnologia, o desenvolvimento de aplicações individuais requer altos investimentos. Passos iniciais no desenvolvimento de padrões de agentes estão sendo encorajados, mas pouca experiência de implementações que os utilizaram foi retornada como referência. Padrões e aplicações são relacionados ao problema "da galinha e do ovo". Padrões precisam pré-datar o desenvolvimento das aplicações, então aquelas aplicações podem ser desenvolvidas de uma maneira consistente, enquanto os exemplos de aplicações precisam existir a fim de que os padrões sejam desenvolvidos. Isto leva a um espiral, em que tanto as aplicações e padrões estão sendo desenvolvidos e aperfeiçoados de forma incremental [MDW99].
- **Ausência de metodologias maduras.** A área de concepção e desenvolvimento de aplicações baseadas em agentes móveis é relativamente recente segundo a perspectiva da engenharia de software. Não existem soluções maduras de metodologias de desenvolvimento de software para este conjunto de aplicações e os métodos e técnicas O.O. (Orientadas a Objetos) existentes para o desenvolvimento de aplicações baseadas em componentes distribuídos não são suficientes para cobrir todos os aspectos de um sistema baseado em agentes móveis.

2.5 Plataformas de Agentes Móveis

Agentes móveis necessitam de um ambiente de execução rodando em todas as máquinas de destino potenciais. Este ambiente de execução é o que chamamos de plataforma de agentes móveis.

Entre outras funcionalidades, uma plataforma de agentes móveis deve fornecer suporte a criação, execução, migração, localização e comunicação dos agentes, como também garantir um ambiente seguro para que eles possam executar [Gro97].

Diversas plataformas de agentes móveis baseadas em Java têm sido desenvolvidas. Entre elas podemos citar:

- **Grasshopper.** Lançada pela IKV++ em 1998, a plataforma Grasshopper é um ambiente de execução e desenvolvimento de agentes móveis construído com base em um ambiente de processamento distribuído, possibilitando assim a integração do paradigma cliente servidor com tecnologia de agentes móveis. Desenvolvida 100% em Java, a plataforma Grasshoper foi a primeira a implementar o primeiro padrão de agentes móveis industrial, o MASIF (*Mobile Agent System Interoperability Facility*),

criado pela OMG (*Object Management Group*) a fim de prover a interoperabilidade entre plataforma de agentes móveis de diferentes fabricantes [IKV01].

- **Aglets.** Criada pelo Laboratório de Pesquisas da IBM do Japão, seu desenvolvimento teve início em 1995, sendo um dos primeiros sistemas de agentes móveis baseados em Java. Seu modelo de agente é baseado no *applet* o que deu origem ao termo Aglet, que é uma fusão das palavras *agent* e *applet* Seus executáveis sempre estiveram disponíveis na forma *freeware*, todavia, apenas recentemente, seu código tornou-se aberto[LO98].
- **Voyager.** Desenvolvido pela ObjectSpace, esta plataforma usa o modelo de objetos da linguagem Java e permite que objetos migrem como agentes na rede. Esta plataforma fornece diversos mecanismos de comunicação, tais como invocação de métodos remotos, *object request broker* e suporte DCOM (*Distributed Component* Object *Model*) [Glas].
- **Concordia.** É uma plataforma de desenvolvimento e gerência de aplicações de agentes móveis que estende qualquer dispositivo suportando Java. Esta plataforma consiste de múltiplos componentes, todos escritos totalmente em Java, que combinados entre si fornecem um ambiente robusto e completo para aplicações distribuídas [MEI98].

2.6 Padrão para a interoperabilidade de plataformas

Várias plataformas de agentes móveis estão sendo desenvolvidas, todavia elas diferem muito no que diz respeito à arquitetura e à implementação. A fim de permitir a interoperabilidade entre as plataformas, alguns aspectos precisam ser padronizados.

Atualmente, o corpo de padronização mais importante na área de agentes móveis é o *Object Management Group* que propôs o padrão *Mobile Agent System Interoperability Facility* (MASIF). O MASIF padroniza as seguintes quatro áreas:

- **Gerência de agentes**. Define operações comuns para a gerência de agentes. Isto permite que o administrador possa localizar agentes, criar um agente dando o nome da classe, suspender a execução do agente, retomar a sua execução ou destruí-lo de uma forma padrão em todas as plataformas.
- **Transferência de agentes**. Define uma infra-estrutura comum, o que permite que os agentes possam migrar por diferentes tipos de sistemas.

- **Nome de sistemas de agentes e nome de agentes.** Define uma sintaxe e uma semântica padrão para os nomes de sistemas de agentes e para os nomes dos agentes. Isto permite que agentes e sistemas sejam capazes de identificar uns aos outros.
- **Tipos de sistemas de agentes e sintaxe de localização.** A sintaxe de localização e o tipo de sistemas de agentes devem ser padronizados para permitir que um agente acesse o tipo do sistema de agentes destino, a fim de verificar se este é capaz de suportá-lo e só em caso afirmativo é que a transferência é realizada.

2.7 Conceitos Básicos

Esta seção define os principais conceitos de agentes móveis tendo como base a plataforma Grasshopper, que foi a escolhida para a implementação do estudo de caso deste trabalho de dissertação.

2.7.1 Ambiente de Agentes Distribuído

Um ambiente de agentes distribuído é composto por regiões, agências, lugares e diferentes tipos de agentes.

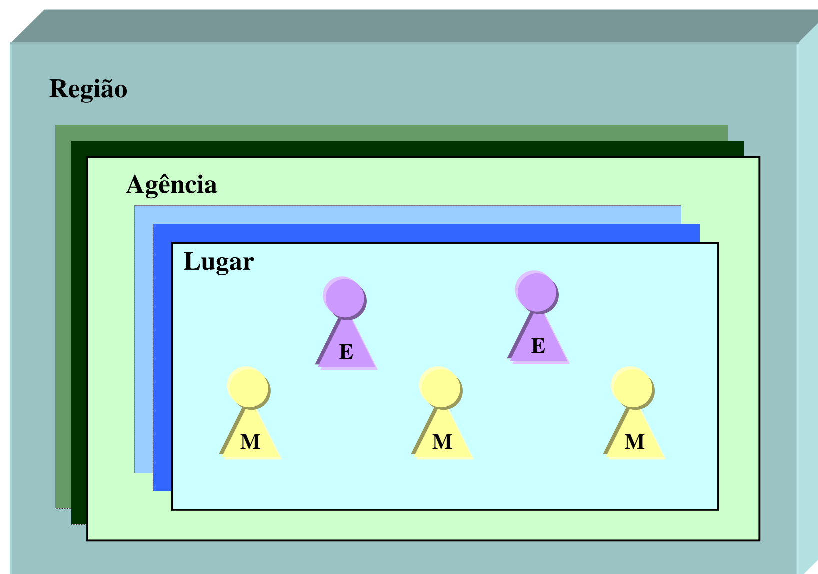

Figura 2.1: Ambiente Distribuído

2.7.2 Agente

Um agente é um programa de computador que age de forma autônoma no lugar de uma pessoa ou uma organização. Atualmente, por questões de portabilidade, muitos agentes são programados em uma linguagem interpretada, tal como Java ou TCL.

Um agente Grasshopper consiste em uma ou mais classes Java, onde a classe principal é denominada de classe agente. Durante sua execução, uma agente Grasshopper é realizado como uma *thread* Java. Geralmente, o comportamento ativo de uma *thread* Java é especificado dentro do seu método `run()`, entretanto este método não é acessível para os programadores dos agentes, devendo o método `live()` ser utilizado para esta finalidade.

2.7.3 Agente Móvel

Um agente móvel é uma entidade de software autônoma que é capaz de migrar entre diferentes localizações da rede e continuar a sua execução do ponto onde parou antes da migração.

2.7.4 Agente Estacionário

Um agente estacionário executa apenas no sistema onde ele iniciou a sua execução. Diferentemente de um agente móvel, ele não possui a habilidade de migrar para um outro sistema, assim se um agente estacionário precisa se comunicar com um agente remoto, ele terá que fazer uso do serviço de transporte de comunicação da plataforma.

2.7.5 Agentes *versus* Objetos

Os conceitos de agentes e objetos costumam ser confundidos devido a algumas similaridades existentes entre eles [Woo99]. Um objeto encapsula algum estado e possue controle sobre ele, de forma que este estado só pode ser acessado ou modificado através de métodos fornecidos pelo próprio objeto. Agentes encapsulam estado de uma forma semelhante. Por outro lado, agentes incorporam uma noção mais forte de autonomia que os objetos, pois diferentemente destes, agentes também encapsulam comportamento, o que faz com que eles tenham controle sobre a execução dos seus métodos, ou seja permite que decidam por eles mesmos se executam ou não uma ação a pedido de outro agente. Além disso, em um sistema multi-agentes, cada agente possui sua própria *thread* de controle o que faz com que eles sejam continuamente ativos, enquanto que objetos são a maior parte do

tempo passivos, tornando-se ativos apenas quando recebem alguma solicitação de outro objeto.

2.7.6 Ciclo de Vida do Agente

A Figura 2.2 apresenta o ciclo de vida de um agente móvel Grasshopper:

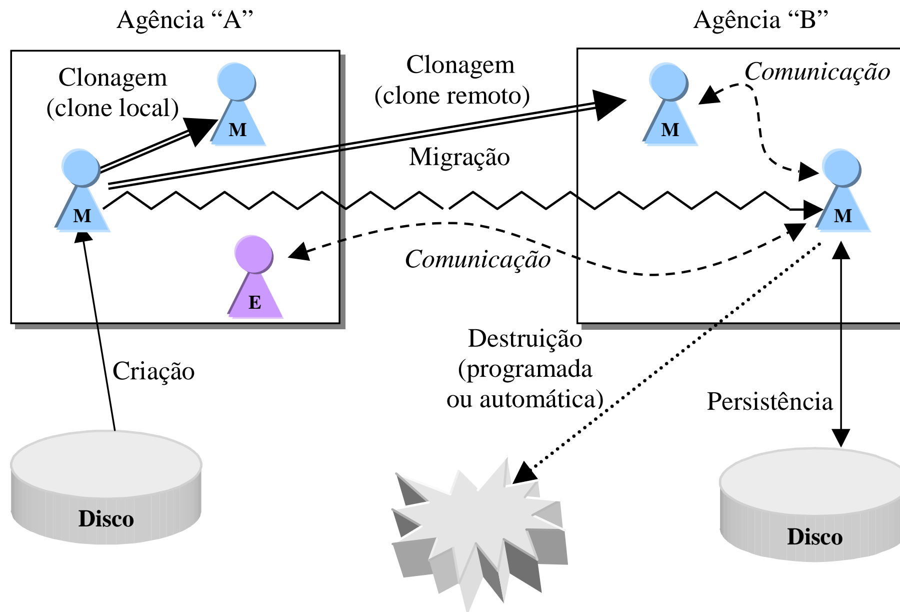

Figura 2.2: Ciclo de vida de um agente móvel

2.7.6.1 Criação

Quando um novo agente é criado ele recebe um identificador único e um conjunto de informações sobre si mesmo, que podem ser acessadas por outros agentes ou por ele mesmo. Todo agente Grasshopper é criado dentro de um lugar mantido por uma agência, esta cria um registro do agente em seu banco de dados interno e o registra na região a qual pertence.

2.7.6.2 Comunicação

Um agente pode se comunicar com outros agentes que podem ser locais ou remotos. O serviço de comunicação Grasshopper permite que a comunicação entre os agentes seja realizada de forma transparente à localização, através do uso de objetos de *proxy*. Isto significa que um agente cliente e um agente servidor podem migrar durante uma seção de comunicação, ficando a cargo do serviço de comunicação localizar o agente servidor e redirecionar as invocações dos métodos para ele.

2.7.6.3 Migração

As plataformas de agentes móveis podem suportar dois tipos de migração: a migração forte e a migração fraca.

Na migração forte, o agente migra levando com ele todo o seu estado de execução (informações de pilha e contador de programa). Após chegar na localização de destino, o agente é instanciado recebendo o mesmo estado de execução, o que possibilita que ele continue a executar a partir do ponto onde havia parado antes da migração.

Já na migração fraca, o agente migra levando com ele apenas o seu estado de dados. O estado de dados de um agente consiste nos valores de variáveis internas serializados antes da migração, transferidos através da rede e fornecidos ao agente na nova localização.

Devido a restrições da JVM, que não permite capturar o estado de execução de um processo ou de uma *thread*, a plataforma Grasshopper, como a grande maioria das outras plataformas baseadas em Java, suporta apenas a migração fraca. Sendo assim, fica a cargo do programador simular a migração forte do agente.

Todo agente Grasshopper deve implementar um método chamado `live()`, que define o comportamento ativo do agente, ou seja, o fluxo de controle que é realizado dentro da *thread* do agente. A fim de simular a migração forte, este método deve ser dividido em blocos de execução diferentes, através de comandos condicionais. Cada bloco deverá cobrir um conjunto de operações que terão que ser executadas em uma única localização. Depois de terminar a execução de um bloco, o agente migra para uma outra localização e executa o próximo bloco.

Na figura abaixo, retirada [IKV01], `estado` é uma variável de instância não transiente definida na classe do agente. Desta forma, `estado` torna-se parte do estado de dados do agente. Pela análise do valor desta variável, o agente determina que bloco deve ser executado. Note que esta variável sempre recebe um novo valor antes que o método `move()` seja invocado.

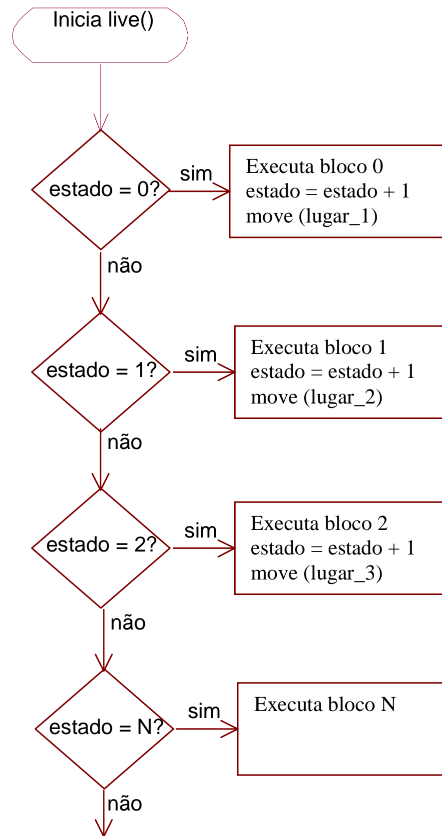

Figura 2.3: Simulação da migração forte

2.7.6.4 Clonagem

Este processo permite que um agente crie uma cópia exata de si mesmo, ou seja, com as mesmas informações internas e com o mesmo estado de execução (este último, apenas nos casos em que a plataforma suporte migração forte). O clone produzido pode ser local ou remoto.

2.7.6.5 Persistência

Ao persistirmos um agente, o seu estado de dados e o seu estado de execução serão armazenados em um sistema de arquivos local da agência hospedeira.

O serviço de persistência oferecido pela plataforma Grasshopper permite que os agentes sejam salvos (informações internas mantidas por eles) periodicamente, à fim de recuperá-los no caso de um *crash* do sistema. O serviço de persistência também é utilizado para desativar os agentes temporariamente. Por restrições semelhantes as da migração, nesta plataforma, apenas o estado de dados do agente é persistido.

2.7.6.6 Destruição

Ocorre quando um agente termina todas as suas atividades e libera todos os recursos que ele está usando. Depois disso seu estado é perdido para sempre [HCMKK00].

2.7.7 Estados do Agente

Durante o seu ciclo de vida um agente Grasshopper pode assumir três estados:

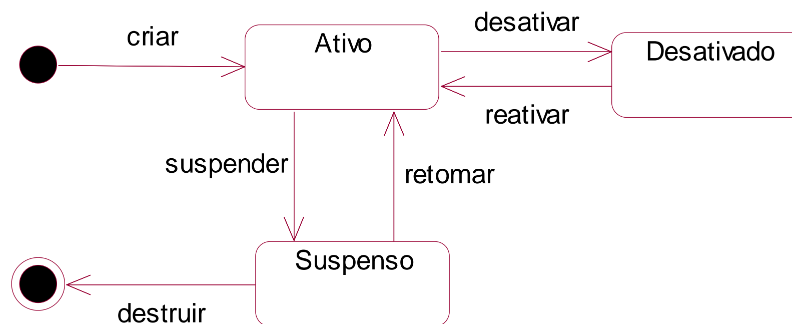

Figura 2.4: Diagrama de estados do agente

- **Ativo:** Imediatamente após a sua criação o agente assume o estado ativo. Neste estado, o agente está realizando a sua tarefa, isto quer dizer que o seu *thread* está executando. Deste estado o agente pode passar para o estado suspenso ou desativado.
- **Suspenso:** Suspender um agente significa suspender seu *thread* e assim interromper a execução da tarefa ativa. Para fazer com que um agente suspenso volte ao estado ativo, a sua execução tem que ser retomada.
- **Desativado:** Quando um agente é desativado, toda informação interna relevante (estado de dados) é permanentemente armazenada. Um agente desativado é reativado automaticamente sempre que um outro componente se comunica com ele invocando algum dos seus métodos.

2.7.8 Agência

A agência corresponde ao ambiente de execução dos agentes. Elas são responsáveis por criar, interpretar, executar, transferir e destruir agentes. Ao menos uma agência deve estar rodando nas máquinas de destino, para que estas sejam capazes de receber e executar agentes.

Uma agência Grasshopper é composta pelo núcleo da agência e por um ou mais lugares.

2.7.8.1 Núcleo da Agência

O núcleo da agência fornece a funcionalidade mínima necessária para que uma agência seja capaz de executar agentes. Os serviços fornecidos pelo núcleo da agência são:

- **Serviço de Comunicação.** Responsável por todas as interações remotas que ocorrem entre componentes Grasshopper, tais como comunicação inter-agente com transparência de localização, transporte de agentes e localização de agentes através do serviço de nomes. Todas as comunicações podem ser realizadas através de conexões do tipo CORBA, RMI, IIOP ou *plain socket*. Opcionalmente conexões RMI e plain socket podem ser protegidas por meio de SSL (*Secure Socket Layer*). O serviço de comunicação suporta mensagens síncronas, assíncronas, comunicação por multicast e invocação dinâmica de métodos.
- **Serviço de Registros.** Mantém o registro de todos os componentes da agência (lugares e agentes hospedados). O serviço de registros de uma agência pode ser conectado ao registro de uma região, está mantém informações sobre todas as agências, lugares e agentes que se encontram no escopo da região.
- **Serviço de Gerência.** Permite que um usuário externo monitore e controle os agentes e lugares de uma agência. Entre outras, as seguintes funcionalidades são fornecidas: criar, remover, suspender, desativar e retomar a execução de agentes e lugares; recuperar informações sobre agentes e lugares específicos; listar todos os agentes residentes em um lugar específico; listar todos os lugares da agência; configurar dados de segurança, rastreabilidade e comunicação.
- **Serviço de Transporte.** Responsável pela migração dos agentes. O serviço de transporte é o responsável pela serialização do agente na agência origem, pela coordenação da transferência dos dados serializados para a agência destino que é realizada pelo Serviço de Comunicação e pela deserialização do agente na máquina destino.
- **Serviço de Segurança.** Grasshopper oferece mecanismos de segurança externa e interna. O mecanismo de segurança externa permite que os agentes (código e estado) sejam criptografados durante a sua migração. E caso seja necessário, a comunicação remota entre componentes Grasshopper distribuídos pode ser protegida através de SSL. Já o mecanismo de segurança interna protege os recursos da agência e os agentes de acessos não autorizados, através de mecanismos de autorização e de autenticação

de usuários. Os mecanismos de segurança interna da plataforma Grasshopper são baseados principalmente em mecanismos de segurança Java.

- **Serviços de Persistência**. Permite salvar agentes e lugares (informações internas mantidas por eles) periodicamente à fim de recuperá-los no caso de um *crash* do sistema. O serviço de persistência também é utilizado para desativar os agentes temporariamente.

2.7.8.2 Lugar

Uma agência pode ter um ou mais lugares e um lugar pode hospedar um ou mais agentes. Um lugar representa um agrupamento lógico das funcionalidades dentro de uma agência. Agências Grasshopper possuem ao menos um lugar, denominado *InformationDesk,* que corresponde ao ponto de entrada padrão para os agentes móveis.

2.7.9 Região

Uma região é uma entidade lógica que agrupa um conjunto de agências que pertencem a uma mesma autoridade como por exemplo, a um domínio de uma empresa ou a um domínio administrativo.

Cada região Grasshopper possui um serviço de registros específico desta plataforma, denominado *region registry*. O *region registry* é responsável por manter informações sobre agências, lugares e agentes da região.

3 MODELO PARA O DESENVOLVIMENTO DE APLICAÇÕES BASEADAS EM AGENTES MÓVEIS

Este capítulo propõe um modelo para o desenvolvimento de aplicações distribuídas baseadas em agentes móveis a partir da investigação e aplicação de teorias, métodos e tecnologias existentes. Uma versão preliminar do modelo a ser proposto já foi apresentada e discutida em [GM01]. As motivações para a proposta deste modelo e trabalhos correlatos são discutidos na Seção 3.1. A Seção 3.2 apresenta uma visão geral do modelo proposto e a Seção 3.3 descreve o estudo de caso utilizado para refinar o modelo e ilustrar a sua descrição nos próximos capítulos. Por fim, considerações finais são apresentadas na Seção 3.4.

3.1 Motivações e Trabalhos Correlatos

Um número de estudos de caso tem demonstrado a aplicabilidade de agentes móveis [VBML99, HB99, VZM00, BM00, KRSW01, GOP02]. No entanto, até agora, a maioria das aplicações baseadas em agentes móveis foram criadas de forma *ad-hoc*, seguindo pouca ou nenhuma metodologia [KRSW01], possivelmente gerando sistemas com níveis de qualidade insatisfatórios e difíceis de manter. Além disso, os métodos e técnicas O.O. (Orientadas a Objetos) existentes para o desenvolvimento de aplicações baseadas em componentes distribuídos não são suficientes para cobrir todos os aspectos de mobilidade na modelagem, projeto e verificação de sistemas baseados em agentes móveis [WJK99, OPB00, MKC01].

A modelagem e projeto de aplicações baseadas em agentes móveis envolvem problemas e decisões não abordados pelo desenvolvimento de software orientado a objetos e pelo desenvolvimento de sistemas cliente-servidor tradicionais. Na fase de análise, o conceito de agentes móveis pode aparecer ao lado de objetos comuns, dependendo do domínio de aplicação. Enquanto que na fase de projeto, padrões de agentes móveis específicos podem ser utilizados para representar soluções a serem implementadas em diferentes plataformas de agentes móveis. Conceitos adicionais terão que ser tratados, tais como: agentes móveis, agentes estacionários, agências e regiões, como também mobilidade e clonagem de agentes.

Metodologias para o desenvolvimento de sistemas multi-agentes têm sido propostas, todavia estas não consideram a propriedade de mobilidade dos agentes ou o fazem de uma forma muito superficial. A maior parte das metodologias propostas focam-se pricipalmente em padrões de interação e cooperação [WJK99, MKC01, GSLM01, OPB00].

Em [WJK99], Wooldrige et al apresenta uma metodologia de análise e projeto orientada a agentes onde estes são modelados através do conceito de papéis que estão associados a responsabilidades, permissões e protocolos de interação. Em [MKC01], Mylopoulos et al propõe a metodologia TROPOS para o de desenvolvimento de sistemas baseados em agentes. Esta metodologia adota o framework de modelagem i* [Yu95], que oferece noções de ator, objetivo e dependência. Tanto a metodologia proposta por Wooldrige et al e por Mylopoulos et al consideram apenas aspectos de interação e cooperação dos agentes.

Em [GSLM01] uma abordagem baseada em aspectos para o projeto de sistemas multiagentes orientados a objetos é apresentada. Neste modelo, o estado e o comportamento básico de um agente é especificado em uma superclasse *Agent* e diferentes tipos de agentes são especificados como subclasses derivadas desta superclasse. As propriedades dos agentes tais como interação, autonomia, adaptação, aprendizagem, colaboração e mobilidade são modeladas através de aspectos que são associados aos tipos de agentes correspondentes. Neste artigo, a ênfase dada a propriedade de mobilidade dos agentes é muito pequena, tendo sido esta citada apenas como uma possível propriedade a ser modelada através de aspectos em diagramas de classes.

Já em [OPB00], Odell et al propõe uma extensão para a UML padrão, chamada Agent UML (AUML), com o objetivo de tornar capaz a representação de todos os aspectos de agentes. Entretanto, as extensões propostas nos diagramas de seqüência, de colaboração, de atividades e de estados são voltadas apenas para protocolos de interação de agentes e o conceito de mobilidade só é considerado nas extensões para os diagramas de implantação (*deployment*).

Abordagens para o desenvolvimento de aplicações baseadas em agentes que tratam de mobilidade de uma forma mais atenciosa já foram apresentadas, contudo limitam-se apenas a atividades específicas do processo de desenvolvimento. Por exemplo, a abordagem proposta em [TOH99] trata apenas da utilização de padrões de agentes móveis na fase de projeto, enquanto que a abordagem apresentada em [KRSW01] está focada apenas na modelagem de agentes móveis utilizando UML. Abordagens mais completas que tratem da análise, projeto, implementação e testes ainda são necessárias como também estudos de casos substanciais que utilizem tais abordagens.

Com o objetivo de contribuir para o uso prático dos conceitos e tecnologia de agentes móveis, este trabalho propõe um modelo para o desenvolvimento de aplicações baseadas em agentes móveis a partir da investigação e aplicação de teorias, métodos e tecnologias existentes. Uma visão geral deste modelo é apresentada na próxima seção.

3.2 O Modelo e seu Ciclo de Vida

O modelo[1] proposto consiste em uma adaptação do processo de desenvolvimento de software apresentado por Larman [Lar02], que é uma versão do processo unificado que utiliza UML e padrões de projeto. Esta adaptação consiste na adição de considerações sobre aspectos relevantes à construção de aplicações baseadas em agentes móveis especificamente nas fases de análise e projeto e na utilização de uma extensão de UML proposta por Klein et al [KRSW01] que permite uma modelagem abstrata adequada dos aspectos de mobilidade. Além disso, este modelo também foi influenciado pelas idéias apresentadas por Tahara et al [TOH99] que, com o objetivo de maximizar o reuso do projeto de aplicações baseadas em agentes móveis, propõe o uso de padrões de acordo com níveis arquiteturais específicos, onde níveis mais altos são independentes de plataformas de agentes específicas. A Figura 3.1 ilustra as principais fases deste modelo:

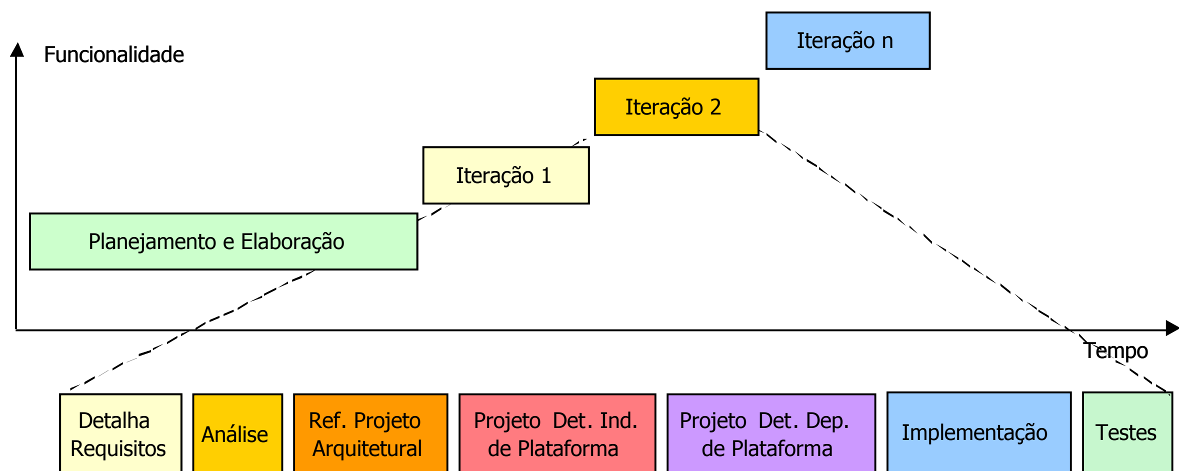

Figura 3.1: Modelo de Desenvolvimento Baseado em Agentes Móveis

A fase de planejamento e elaboração tem o objetivo de delimitar o escopo do projeto, conhecer o domínio do problema, levantar requisitos funcionais e não funcionais do sistema, estabelecer uma arquitetura inicial e priorizar as funcionalidades e distribuí-las entre as interações. A execução das atividades desta fase ocorrem conforme sugerido por Larman [Lar02], onde nenhuma consideração relacionada a agentes móveis foi acrescentada. Os principais artefatos gerados durante esta fase são: plano de desenvolvimento de software,

---

[1] Por modelo de desenvolvimento entendemos um conjunto de atividades a ser executada para a produção de um software, onde para cada atividade existem métodos e notações associadas.

relatório de investigação preliminar (*business case*), especificação de requisitos, glossário, projeto arquitetural inicial (a ser refinado na fase de projeto), modelo conceitual inicial (refinado na fase de análise) e protótipo (opcional).

A fase de construção, que engloba as fases de análise, projeto, implementação, testes e transição, é iterativa e incremental. Isto é, possui várias iterações no tempo e em cada iteração uma nova versão incrementada do sistema é gerada. Cada iteração consiste em um ciclo de desenvolvimento completo.

A fase de análise tem o objetivo de gerar um modelo de alto nível do domínio do problema que deve ser voltada para o entendimento do usuário, mas que também serão utilizados pelos projetistas do sistema servindo de base para a execução das fases de projeto e testes. Na fase de análise o conceito de agentes pode começar a ser introduzido dentro das próprias especificações de caso de uso, assumindo o papel do sistema, desde que eles sejam intrínsecos ao domínio do problema. Este modelo assume que um sistema é composto não só por objetos, mas também por agentes e que estes são capazes de interagir entre si, de forma que ambos podem figurar lado a lado no modelo conceitual.

A fase de projeto consiste em uma extensão do modelo de análise visando especificar em detalhes como a solução será implementada. A fim de permitir o uso efetivo de padrões de projeto de agentes móveis e promover o reuso, a fase de projeto é dividida em três subfases: projeto arquitetural, projeto independente de plataforma e projeto dependente de plataforma. Esta divisão é baseada nas idéias apresentadas por Tahara et al [TOH99], onde os sistemas baseados em agentes são projetados de acordo com os seguintes níveis: macroarquitetura – projeto de alto nível independente de plataforma – e microarquitetura – projeto detalhado e comportamento de agentes especializado em uma plataforma específica. Em nosso modelo, as fases de projeto arquitetural e de projeto dependente de plataforma correspondem aos níveis de projeto de macroarquitetura e de microarquitetura propostos por Tahara et al. A principal diferença entre os dois modelos é que nós incluímos a fase de projeto independente de plataforma que trata de todo o projeto lógico da aplicação sem entrar em detalhes relacionados a uma plataforma de agentes móveis específica. Esta divisão foi motivada pela possibilidade de reuso do projeto, pois caso haja alteração na escolha da plataforma de agentes móveis apenas a segunda fase do projeto detalhado precisará ser refeita. A idéia de gerar um projeto independente e outro dependente de plataforma não é excluiva do nosso modelo, ela também pode ser encontrada na abordagem para especificação de sistemas e interoperabilidade baseada no uso de modelos formais M.D.A (*Model-Driven Architecture*) proposta pela O.M.G. (*Object Management Group*) [Poo2001]. Em M.D.A., modelos

independentes de plataforma são inicialmente expressos em uma linguagem de modelagem independente de plataforma, tal como UML, e posteriormente mapeados para uma plataforma ou linguagem de implementação (ex.: Java, XML, SOAP) através do uso de regras formais gerando modelos dependentes de plataforma.

Um outro diferencial da fase de projeto detalhado é que os diagramas de classe e os diagramas de interação, através da utilização da extensão de UML proposta por Klein et al [KRSW01], são capazes de representar regiões, agências, agentes estáticos, agentes móveis, como também mobilidade e clonagem de agentes.

A fase de implementação consiste na codificação das especificações detalhadas na fase de projeto dependente de plataforma. E a fase de testes pode incluir testes feitos pelo próprio programador (testes de unidade), testes funcionais, testes de sistema, entre outros.

3.3 Estudo de Caso

A fim de refinar e analisar a viabilidade do modelo proposto, realizamos um estudo de caso através do desenvolvimento de uma aplicação para a Internet sugerida por Cardelli em [Car99]: Sistema de Apoio às Atividades de Comitês de Programa em Conferências. Esta aplicação gerencia as atividades de um comitê de programa de uma conferência, tais como a submissão de artigos, o processo de revisão e a notificação para os autores sobre a aceitação ou rejeição dos artigos.

É requisito desta aplicação que muitas interações aconteçam na ausência de conectividade. É improvável que todos os membros do comitê estejam conectados permanentemente durante todo o período de revisão e resolução de conflitos, que em geral, duram no mínimo um mês. Assim, revisões de artigos são feitas muitas vezes em aviões, em salas de espera, em lanchonetes, etc. Enquanto um laptop pode ser facilmente levado para estes ambientes, a conectividade contínua ainda está longe de ser alcançada. Além disso, o preenchimento do formulário requer verificação semântica que é melhor realizada enquanto o formulário está sendo preenchido. Então, formulários ativos são necessários mesmo durante a operação *off-line*. Um outro requisito desta aplicação é que os formulários de revisão podem ser encaminhados para uma cadeia de revisores e devem encontrar o caminho de volta para o coordenador de programa, permitindo que ele esteja livre de acompanhar todas as atividades envolvidas no repasse dos formulários de revisão e da recuperação deles. Além disso, o sistema deverá tratar de domínios administrativos múltiplos, visto que os membros do comitê estão dispersos geograficamente.

Devido a restrições de tempo apenas uma iteração do modelo foi executada abordando os seguintes casos de uso:

- Iniciar Processo de Revisão de um Artigo
- Redirecionar Formulário de Revisão
- Revisar Artigo
- Aprovar Revisão

Estes casos de uso foram selecionados por serem os que englobam a maior parte da mobilidade do sistema, o que nos possibilitou testar a arquitetura do sistema, bem como refinar o modelo que está sendo proposto, através da sua aplicação durante o desenvolvimento desta iteração.

Para implementarmos tais casos de uso levaremos em conta que todos os dados que seriam fornecidos ao sistema (ex.: datas importantes da conferência, formulário de submissão e membros do comitê), através de outros casos de uso, já foram armazenados no banco de dados.

3.4 Considerações Finais

Este capítulo apresentou uma visão geral do modelo proposto para o desenvolvimento de aplicações baseadas em agentes móveis. Este modelo não sugere um processo totalmente inovador, mas consiste na combinação de técnicas e métodos amplamente utilizados por projetistas de software, tais como o processo unificado, UML e padrões de projeto, o que facilita o seu aprendizado e adoção.

Nos Capítulos 4, 5, 6 e 7, respectivamente, as fases de análise, projeto arquitetural, projeto detalhado independente de plataforma e projeto detalhado dependente de plataforma serão explanadas e ilustradas pelo estudo de caso apresentado na Seção 3.3 deste capítulo. Cada fase do modelo será descrita através dos seguintes atributos: objetivo principal, artefatos de entrada, tarefas a serem realizadas, artefatos e critérios de validação.

4 FASE DE ANÁLISE

Este capítulo apresenta a fase de análise do modelo proposto, ressaltando como o conceito de agentes pode ser introduzido gradualmente nas fases iniciais de um processo de desenvolvimento. Durante as especificações dos casos de uso, o conceito de agentes pode ser utilizado assumindo o papel do sistema, desde que eles representem abstrações de conceitos do mundo real. A afirmação anterior, também se aplica aos diagramas de sistemas gerados para cada caso de uso. O modelo proposto assume que um sistema é composto não só por objetos, mas também por agentes e que estes são capazes de interagir entre si, de forma que ambos podem figurar lado a lado no modelo conceitual. É importante ressaltar que a presença de conceitos de agentes, como abstrações do mundo real, na fase de análise não deve necessariamente direcionar a uma solução orientada a agentes (entidades de software).

A seguir, a fase de análise será descrita através do seu objetivo, dos seus artefatos de entrada, das tarefas a serem realizadas durante a sua execução, dos artefatos a serem produzidos e dos critérios de validação.

4.1 Objetivo

Gerar um modelo de alto nível do domínio do problema de acordo com a iteração corrente. Os artefatos produzidos nesta fase devem ser voltados para o entendimento do usuário, mas também serão utilizados pelos projetistas do sistema servindo de base para a execução das fases de projeto e testes.

4.2 Entradas

Os seguintes artefatos gerados durante a fase de Planejamento e Elaboração auxiliarão na execução desta fase:

- Documento de investigação preliminar (*business case*)
- Documento de especificação de requisitos
    - Requisitos funcionais (Modelo de use case de alto nível)
    - Requisitos não funcionais
- Plano de Projeto

4.3 Tarefas

As principais tarefas a serem realizadas nesta fase são:

- Expandir os casos de uso que farão parte da iteração
- Elaborar modelo conceitual
- Definir glossário
- Definir o comportamento do sistema:
    - Criar diagramas de seqüência do sistema
    - Criar diagrama de atividades
    - Criar contratos de operação do sistema

4.4 Artefatos

Os artefatos gerados nesta fase são similares aos usualmente produzidos na análise orientada a objetos, com a exceção de que os conceitos de agentes são introduzidos gradualmente à medida que as especificações vão sendo mais detalhadas.

4.4.1 Casos de Uso Expandidos

Os requisitos funcionais descrevem o que o sistema deve fazer e podem ser explorados e documentados através de casos de uso, que é uma técnica usada para descrever os processos do domínio do problema.

Nesta fase, os casos de uso essenciais de alto nível (definidos na fase de planejamento e elaboração) que farão parte desta iteração serão expandidos. Aqui, o conceito de agentes pode começar a ser introduzido assumindo o papel do sistema, desde que eles sejam intrínsecos ao domínio do problema, ou seja, desde que eles representem abstrações de conceitos do mundo real, tais como pessoas ou objetos ativos e móveis.

Um caso de uso expandido mostra mais detalhes que um de alto nível e ajuda a entender de uma forma mais profunda os processos e os requisitos do sistema. O que difere o caso de uso expandido do de alto nível é a seção *seqüência típica dos eventos* que descreve em detalhes a interação entre os atores e o sistema. Uma particularidade desta seção é que ela descreve apenas a seqüência de eventos mais comum de um processo do sistema, situações alternativas ou exceções são descritas na seção final, *seqüências alternativas.*

Selecionamos quatro casos de uso para o desenvolvimento da primeira iteração do Sistema de Apoio a Comitês de Programa em Conferências: iniciar processo de revisão de um

artigo, redirecionar formulário de revisão, revisar artigo e aprovar revisão. Observe o diagrama de caso de uso da Figura 4.1:

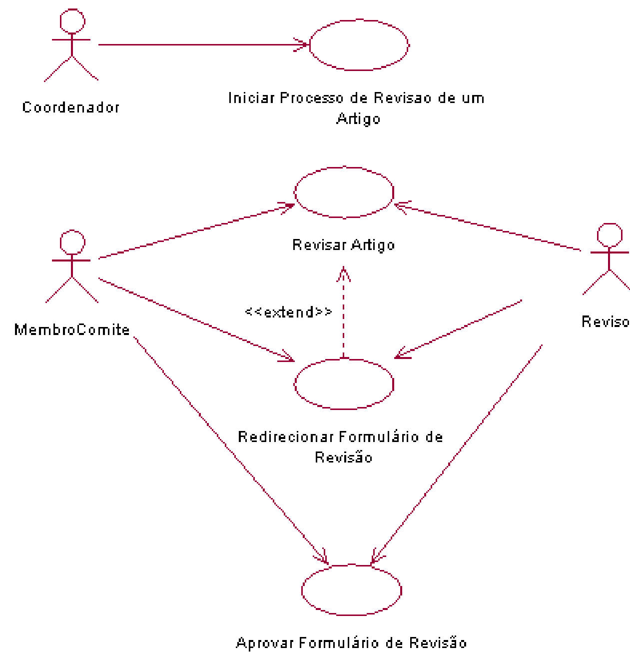

Figura 4.1: Diagrama de caso de uso parcial para o Sistema de Apoio a Comitês de Programa em Conferências

As especificações destes casos de uso estão expandidas a seguir. Note que os conceitos de agente estático (agente coordenador) e agente móvel (agente formulário de revisão) já aparecem nas especificações dos casos de uso. Os agentes fazem parte do sistema e só devem figurar nas especificações dos casos de uso quando eles interagirem diretamente com os atores do sistema.

O agente coordenador interage diretamente com o coordenador de programa recebendo as suas solicitações e age em nome dele controlando todo o processo de revisão. Desta forma, a única responsabilidade do coordenador de programa é encaminhar os formulários de revisões dos artigos para os membros do comitê. E o agente formulário de revisão é o responsável por coletar as informações da revisão, levá-las de um revisor a outro e retornar com elas para o coordenador de programa. Assim, o agente formulário de revisão interage diretamente com os membros de comitê e com os revisores, que podem redirecioná-lo para outro revisor ou entrar com dados da revisão.

Use Case:	**Iniciar processo de revisão de um artigo**
Atores:	Coordenador do programa

Agentes: Agente Coordenador, Agente Formulário de Revisão

Descrição: O coordenador do programa seleciona um artigo, escolhe um membro de comitê que será o responsável por sua revisão e indica a quantidade de cópias de formulários de revisão que deverão ser entregues a ele. As cópias entregues ao membro do comitê deverão ser redirecionadas para outros revisores.

Seqüência típica de eventos

Ação do ator	Resposta do sistema
1. O coordenador do programa solicita ao sistema que entregue N cópias de formulários de revisão de um artigo para um determinado membro de comitê. Cada formulário de revisão deverá conter uma cópia do arquivo do artigo.	2. O sistema (agente coordenador) gera o agente formulário de revisão.
	3. O agente formulário de revisão migra para a máquina do membro do comitê.
	4. O agente formulário de revisão gera as cópias solicitadas.

Seqüências alternativas:

Linha 4: O agente formulário de revisão não consegue migrar para a máquina do membro do comitê e exibe uma mensagem de erro.

Caso de uso: **Redirecionar formulário de revisão**

Atores: Membro do Comitê ou Revisor

Agentes: Agente Formulário de Revisão

Descrição: O membro do comitê ou o revisor recebe o agente formulário de revisão e decide redirecioná-lo. O membro do comitê informa os dados da agência do revisor para a Qual ele deve migrar e se deseja que o agente retorne a sua máquina após o formulário de revisão ter sido completado (a intenção é que cada revisor possa verificar a revisão dos sub-revisores). O agente formulário de revisão migra para a agência indicada.

Seqüência típica de eventos

Ação do ator	Resposta do sistema
1. O membro do comitê ou o revisor recebe o agente formulário de revisão e decide redirecioná-lo.	
2. O membro do comitê ou o revisor informa nome da agência do revisor, o endereço da região na qual ela está inscrita e se o agente deve retornar quando o processo de revisão tiver sido completado.	3. O agente formulário de revisão migra para a agência indicada.

Seqüências alternativas:

Linha 1: O membro do comitê ou o revisor ativa o formulário de revisão, que estava desativado, para então poder redirecioná-lo.

Linha 2: O membro do comitê ou o revisor desativa o formulário e deixa para redirecioná-lo depois.

Linha 3: O agente formulário de revisão não consegue migrar e exibe uma mensagem de erro.

Use Case: **Revisar artigo**

Atores: Membro do Comitê ou Revisor

Agentes: Agente Formulário de Revisão

Descrição: O membro do comitê ou o revisor recebe o agente formulário de revisão contendo o artigo submetido em anexo. O membro do comitê ou o revisor lê o artigo e preenche todos os campos do formulário de revisão. O agente formulário de revisão volta para o coordenador de programa.

Seqüência típica de eventos

Ação do ator	Resposta do sistema
1. O membro do comitê ou o revisor recebe o agente formulário de revisão e decide revisá-lo.	
2. O membro do comitê ou o revisor solicita a extração do arquivo do artigo em	

um diretório em sua máquina.	
3. Após ter lido o artigo, o membro do comitê ou o revisor preenche os campos do formulário de revisão.	
4. O revisor finaliza o processo de revisão.	5. O agente formulário de revisão retorna para a máquina do coordenador.

Seqüências alternativas:

Linha 1: O membro do comitê ou o revisor ativa o formulário de revisão, que estava desativado, para completar o processo de revisão do artigo.

Linha 2, 3, 4: O membro do comitê ou o revisor desativa o formulário e deixa para completar o processo de revisão do artigo depois.

Linha 2, 3, 4: O membro do comitê ou o revisor redireciona o formulário para outro revisor.

Linha 4: Algum campo foi preenchido de forma incorreta ou foi deixado em branco. Indica erro.

Linha 5: O agente formulário de revisão volta para a máquina do último remetente (membro do comitê ou revisor) que solicitou o seu retorno.

Linha 5: O agente formulário de revisão não consegue migrar e exibe uma mensagem de erro.

Use Case:	**Aprovar formulário de revisão**
Atores:	Membro do Comitê ou Revisor
Agentes:	Agente Formulário de Revisão
Descrição:	O membro do comitê ou o revisor recebe o agente formulário de revisão com os dados preenchidos. Checa as informações, as altera se necessário e aprova a revisão.

Seqüência típica de eventos

Ação do ator	***Resposta do sistema***
1. O membro do comitê ou o revisor recebe o agente formulário de revisão com os dados preenchidos.	
2. O membro do comitê ou o revisor checa	3. O agente formulário de revisão retorna

as informações e aprova a revisão.	para a máquina do coordenador de programa.

Seqüências alternativas:

Linha 1: O revisor ativa o formulário de revisão, que estava desativado, para completar o processo de aprovação de revisão.

Linha 2: Antes de aprovar a revisão, o revisor pode alterar os dados da revisão.

Linha 2: O revisor desativa o formulário e deixa para completar o processo de aprovação depois.

Linha 3: O agente formulário de revisão volta para a máquina do último remetente (membro do comitê ou revisor) que solicitou o seu retorno.

Linha 3: O agente formulário de revisão não consegue migrar e exibe uma mensagem de erro.

4.4.2 Modelo Conceitual

O modelo conceitual ilustra os conceitos importantes do domínio do problema, suas associações e atributos. Sua criação é dependente dos casos de uso e de outros documentos, onde os conceitos podem ser identificados. Não necessariamente o modelo conceitual deve ser criado após os casos de uso, eles podem ser criados em paralelo.

Este modelo assume que um sistema é composto não só por conceitos de objetos, mas também por conceitos de agentes e que estes são capazes de interagir entre si, de forma que ambos podem figurar lado a lado no modelo conceitual. É importante ressaltar que os conceitos levantados aqui são conceitos do domínio do problema e não entidades de software, desta forma a solução não deve necessariamente ser direcionada a utilização da tecnologia de agentes, podendo estes conceitos serem implementados de outra forma.

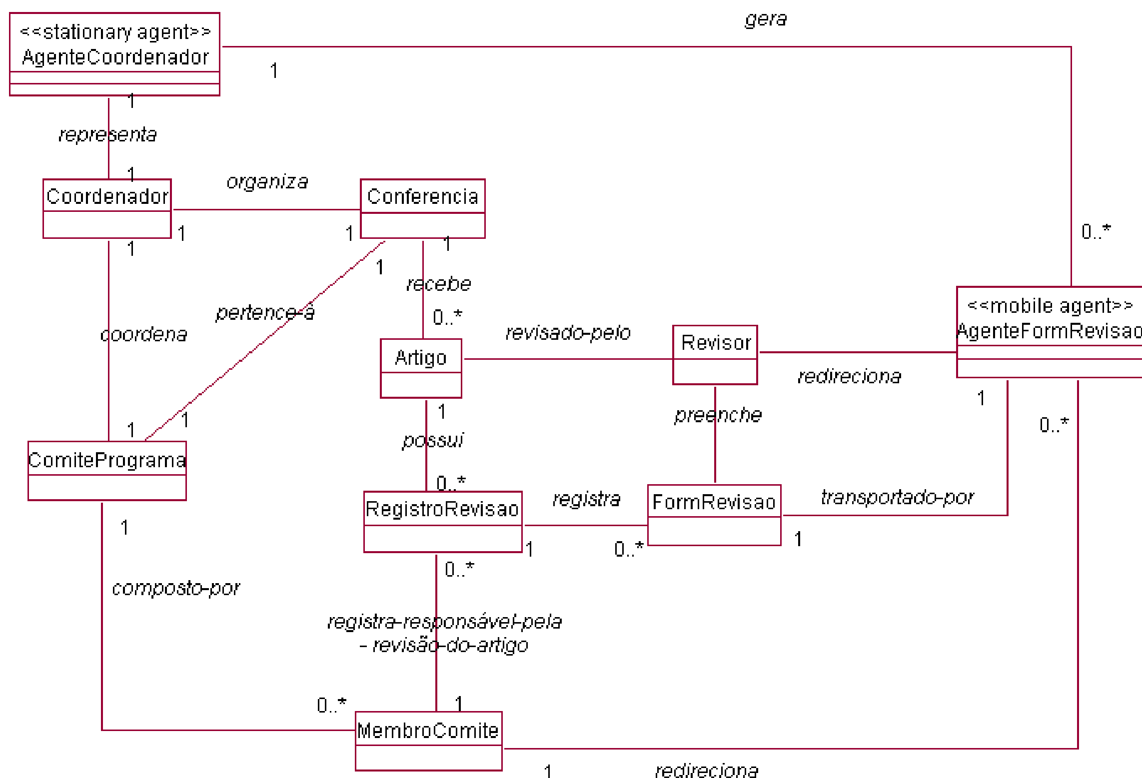

Figura 4.2: Modelo conceitual parcial do estudo de caso

A Figura 4.2 apresenta o modelo conceitual parcial para o domínio de Conferências, onde os atributos foram omitidos por questões de espaço. Para diferenciar os conceitos de agentes dos conceitos de objetos, podemos utilizar estereótipos. Com este objetivo, Klein et al [KRSW01] propôs a utilização do estereótipo <<*mobile agent*>>, derivado do metamodelo *class* de UML, que especifica o conceito de um agente móvel. Todavia, ele não sugeriu nenhum estereótipo que identificasse o conceito de agente estacionário, para esta finalidade propomos a utilização do estereótipo <<*stationary agent*>>.

Ao migrar, o *AgenteFormRevisao* transporta com ele apenas os dados relevantes à execução de sua tarefa. Dados sobre a conferência e sobre o artigo terão que ser exibidos para o membro do comitê e para os revisores. Ele também precisa dos dados do registro de revisão a fim de saber para qual membro do comitê ele deve migrar e quantas cópias de si próprio ele deve criar. O *AgenteFormRevisao* também possui um itinerário que armazena os destinos para os quais o agente terá que migrar. O modelo conceitual, ilustrado na Figura 4.3, mostra os detalhes do *AgenteFormRevisao*.

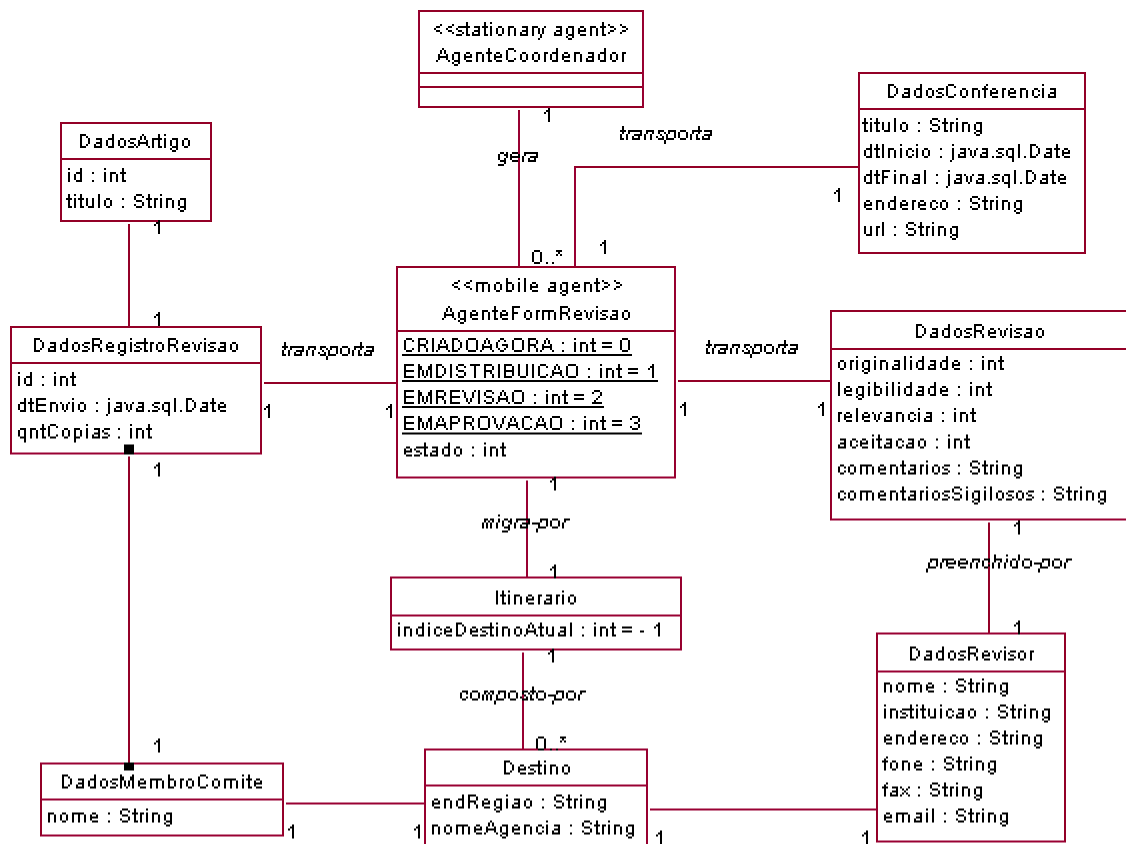

Figura 4.3: Modelo conceitual do AgenteFormRevisao

4.4.3 Glossário

A definição de termos do domínio do problema é importante, pois reduz o risco de falhas de entendimento, garantindo uma comunicação clara entre analistas e clientes.

Um glossário é um documento simples que define termos devendo ser criado inicialmente na fase de planejamento e elaboração, quando os termos começam a ser identificados, e continuar a ser refinado durante todo o ciclo de desenvolvimento, quando surgem novos termos. Ele geralmente é elaborado em paralelo com a especificação de requisitos, casos de uso e modelo conceitual.

Não existe um formato oficial para um glossário, mas segue abaixo um exemplo de um glossário incompleto para o Sistema de Apoio a Comitês de Programa em Conferências:

Termo	Categoria	Comentário
AgenteCoordenador	Conceito	Agente estacionário que age em nome do coordenador de programa. Ele é o responsável por gerenciar todas as atividades do sistema.

AgenteFormRevisao	Conceito	Agente móvel responsável por transportar os dados da revisão. Este agente irá migrar entre as máquinas do coordenador de programa, do membro do comitê e dos revisores.
Artigo	Conceito	Documento submetido à avaliação do comitê de programa.
ComitePrograma	Conceito	Conjunto de pessoas responsáveis pelas revisões dos artigos.
Conferencia	Conceito	Evento para o qual os artigos são submetidos.
Coordenador	Conceito	Coordenador de programa, pessoa que coordena todas as atividades de um comitê de programa.
Estado	Atributo	Atributo do *AgenteFormRevisao*. Representa o estado de execução do agente e pode assumir os seguintes valores: CRIADOAGORA – estado assumido logo após a sua criação. EMCLONAGEM – estado assumido antes de migrar da agência do coordenador de programa para a agência do membro de comitê. EMDISTRIBUICAO – estado assumido quando está aguardando ser redirecionado pelo membro do comitê para um revisor. EMREVISAO – estado assumido após ter sido redirecionado pelo membro de comitê para um revisor. EMAPROVACAO – estado assumido após ter sido revisado.
FormRevisao	Conceito	Formulário a ser preenchido por um revisor com informações sobre a revisão de um artigo.
MembroComite	Conceito	Pessoa que faz parte do comitê de programa. Membros de comitê de programa são responsáveis pelas revisões dos artigos e cabem a eles o redirecionamento dos formulários de

		revisão para outros revisores.
Itinerario	Conceito	Armazena o caminho do agente.
Revisor	Conceito	Pessoa responsável por revisar um artigo ou redirecioná-lo para outro revisor.

4.4.4 Diagramas de Seqüência do Sistema

Antes de partir para o projeto lógico de como a aplicação de software trabalhará, é necessário investigar e definir seu comportamento como uma caixa preta. O comportamento do sistema é uma descrição **do que** o sistema faz, sem explicar **como** ele faz.

Os diagramas de seqüência são elaborados durante a fase de análise de um ciclo de desenvolvimento a partir dos casos de uso.

Um diagrama de seqüência do sistema mostra, para um cenário particular de um caso de uso, os eventos gerados pelos atores externos, sua ordem, além de eventos envolvendo outros sistemas. Todos os sistemas são tratados como caixas pretas, dando-se ênfase aos eventos que partem dos atores para o sistema. Agentes que interagem diretamente com os atores podem representar o sistema.

A Figura 4.4 ilustra o diagrama de seqüência para o curso típico do caso de uso Revisar artigo. Este diagrama indica que o revisor é apenas um ator para o sistema e que ele gera os eventos do sistema *extrairArquivo, entrarDadosRevisao, fimRevisao*. Além disso, o sistema está sendo representado pelo *AgenteFormRevisao*, que é com quem o ator interage de fato.

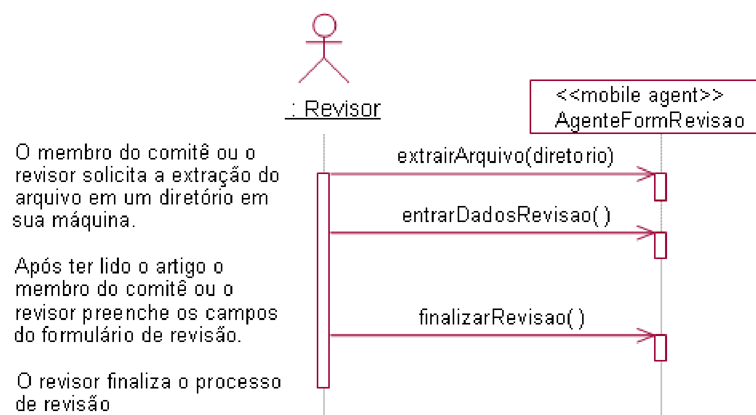

Figura 4.4: Diagrama de seqüência do caso de uso Revisar artigo

Os diagramas de seqüência dos outros casos de uso são relativamente simples e por este motivo não serão ilustrados.

4.4.5 Diagrama de Atividades

A seqüência de atividades de um caso de uso consiste em uma seqüência básica e em uma ou mais seqüências alternativas que podem ser representadas graficamente através de um diagrama de atividades. As Figuras 4.5, 4.6, 4.7, 4.8 apresentam os diagramas de atividades gerados para os casos de uso desta iteração.

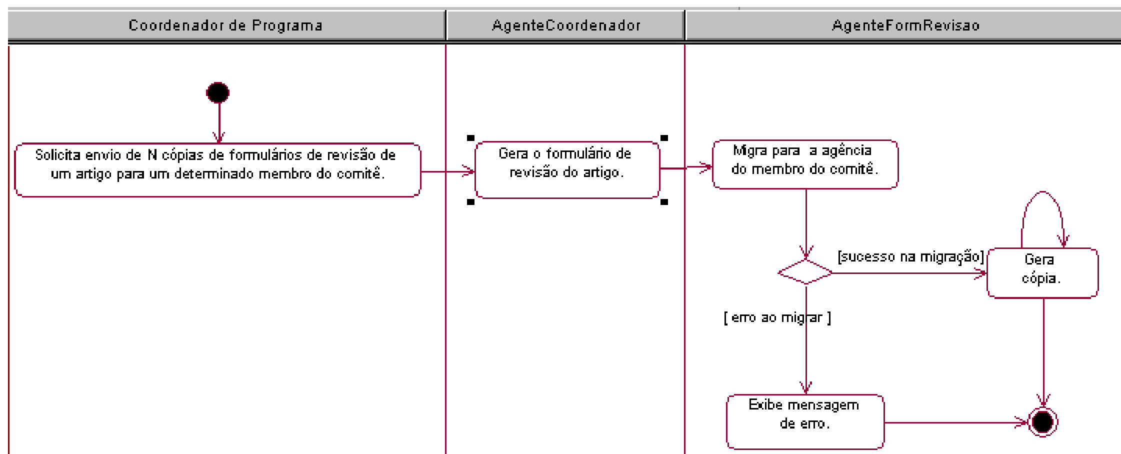

Figura 4.5: Diagrama de atividades do caso de uso "Iniciar processo de revisão de um artigo".

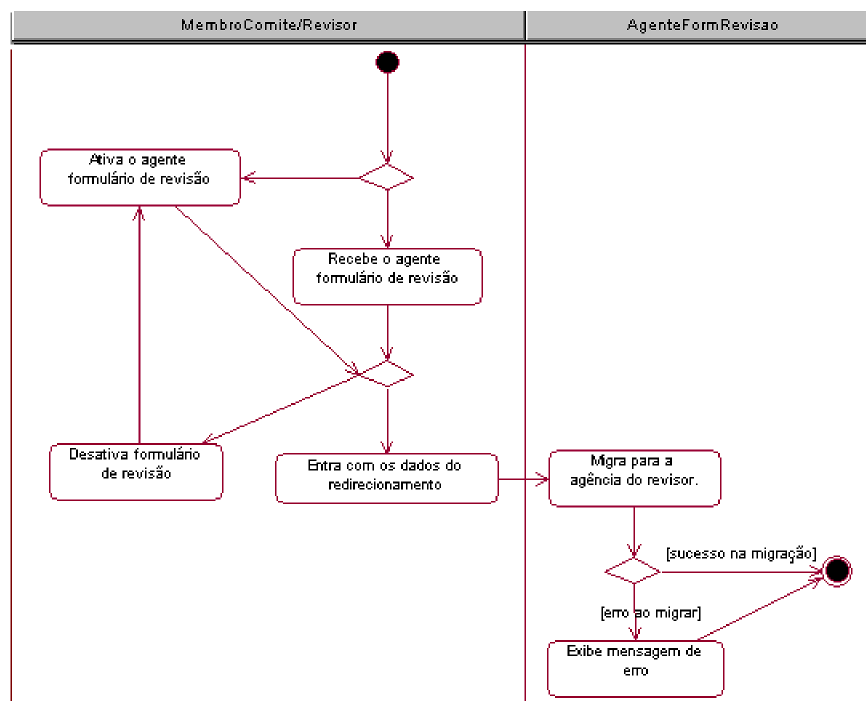

Figura 4.6: Diagrama de atividades do caso de uso "Redirecionar Formulário de Revisão".

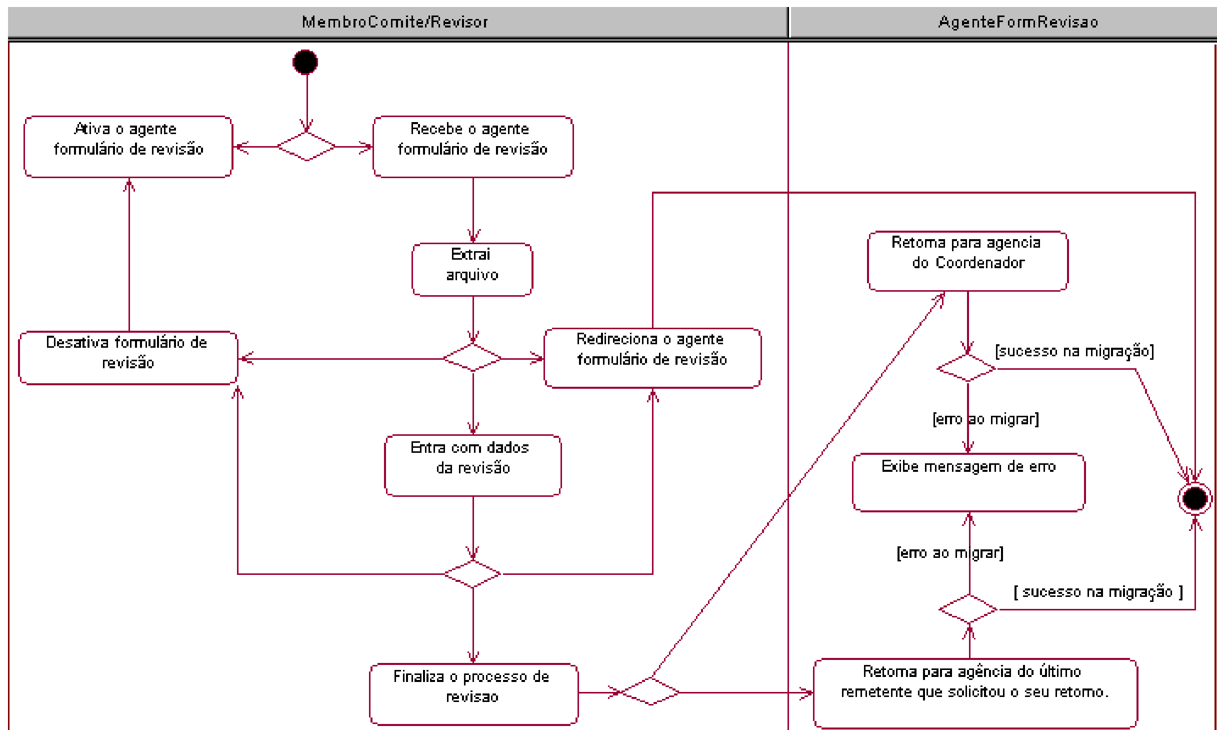

Figura 4.7: Diagrama de atividades do caso de uso "Revisar artigo"

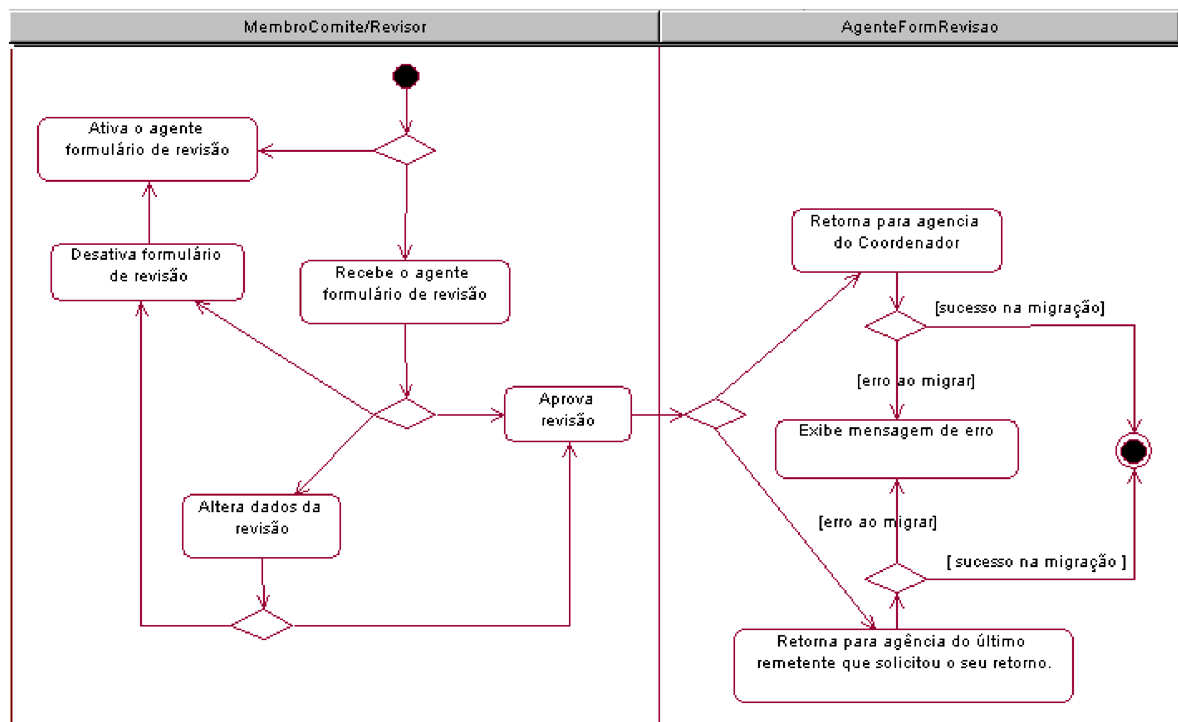

Figura 4.8: Diagrama de atividades do caso de uso "Aprovar revisão"

4.4.6 Contratos

Um diagrama de seqüências do sistema mostra os eventos gerados pelo ator externo, mas não menciona as funcionalidades associadas às operações de sistema invocadas, que são detalhes importantes do comportamento do sistema.

Em geral, um contrato é um documento que descreve o que uma operação promete cumprir. Eles descrevem os efeitos das operações no sistema, ou seja, o que muda quando uma operação é invocada.

Os contratos são elaborados durante a fase de análise de um ciclo de desenvolvimento. Eles são dependentes dos diagramas de seqüência do sistema, pois um contrato deve ser construído para cada operação do sistema.

Contratos são normalmente expressos em termos de pré e pós-condições, que descrevem as mudanças do sistema. As pré-condições são suposições sobre o estado do sistema antes da execução da operação. Enquanto que as pós-condições mencionam as alterações do estado do sistema como resultado da execução da operação. Segundo Larman [Lar02], as pós-condições podem ser descobertas nas seguintes categorias:

- Criação e destruição de instâncias
- Modificação de atributos
- Associações formadas e quebradas

Porém, para aplicações baseadas em agentes móveis, nós identificamos ainda outras categorias:

- Migração (Quando o agente migra para outra localização)
- Desativação (Quando o agente é desativado)
- Ativação (Quando o agente é ativado)

Uma descrição de uma seção de contrato é mostrada no esquema a seguir. Nem todas as seções são necessárias, embora as seções Responsabilidades e Pós-condições sejam recomendadas.

Contrato	
Nome:	Nome da operação e parâmetros.
Responsabilidades:	Uma descrição informal das responsabilidades desta operação.
Referências:	Números de referências das funções do sistema, casos de uso etc.

Tipo:	Nome do tipo. Pode ser Sistema, Classe de *software* ou Interface, dependendo do tipo de contrato que se está elaborando.
Exceções:	Casos excepcionais.
Saída:	Indica informação que sai do sistema (mensagens ou registros), sem incluir a interface com o usuário.
Pré-condições:	Suposições sobre o estado do sistema antes da execução da operação.
Pós-Condições:	As pós-condições mencionam as alterações do estado do sistema como resultado da execução da operação.

Para o nosso estudo de caso, as seguintes operações de sistema foram identificadas:

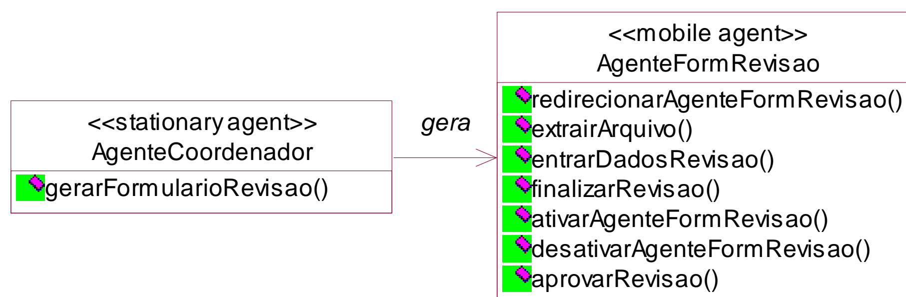

Figura 4.9: Operações do sistema

O contrato de cada uma destas operações é apresentado a seguir:

Contrato	
Nome:	**gerarFormRevisao(idArtigo, idMembroComite, qntCopias)**
Responsabilidades:	Gerar o formulário de revisão com o arquivo do artigo a ser revisado em anexo e entregar para o membro do comitê *qntCopias* cópias do formulário.
Referências	Caso de uso: Gerar Formulário de Revisão
Tipo:	Sistema
Exceções:	Se o identificador do artigo não é conhecido pelo sistema, indicar erro. Se o identificador do membro do comitê não for conhecido pelo sistema, indicar erro.

	Se qntCopias não possuir valor válido, indicar erro. Se a agência do membro do comitê estiver desconectada, o *AgenteFormRevisao* (agente móvel) deverá tentar migrar após um certo tempo. Se após um certo número de tentativas o *AgenteFormRevisao* não conseguir migrar, indicar erro.
Saída:	Exibe mensagem informando que o formulário de revisão foi gerado com sucesso e que [*qntCopias*] cópias dele serão entregues ao membro do comitê.
Pré-condições:	O identificador do artigo deve ser conhecido pelo sistema. O identificador do membro do comitê deve ser conhecido pelo sistema. O parâmetro *qntCopias* deve possuir um valor inteiro maior que 0. A agência do revisor deve estar conectada.
Pós-condições:	O objeto *RegistroRevisao* foi criado (*criação de instância*) e associado ao *Artigo* (*formação de associação*). *RegistroRevisao.id* recebeu um número seqüencial (*mudança de atributo*). *RegistroRevisao.qntCopias* recebeu o valor de *qntCopias* (*mudança de atributo*). *RegistroRevisao.dtEnvio* recebeu data atual do sistema. Um **agente móvel** *AgenteFormRevisao* foi criado (*criação de instância*) e migrou para a agência do *MembroComite* (*migração*). *AgenteFormRevisao.estado* recebeu *EMDISTRIBUICAO* (*mudança de atributo*). *AgenteFormRevisao* criou [*qntCopias*–1] clones de si próprio (*criação de instância)* na agência do *MembroComite.* O objeto *DadosConferencia* foi criado (*criação de instância*) e associado ao *AgenteFormRevisao* (*formação de associação*). *DadosConferencia.titulo* recebeu *Conferencia.titulo* (*mudança de atributo*). *DadosConferencia.dtInicio* recebeu *Conferencia.dtInicio*

	(*mudança de atributo*). *DadosConferencia.dtFinal* recebeu *Conferencia.dtFinal* (*mudança de atributo*). *DadosConferencia.endereco* recebeu *Conferencia.endereco* (*mudança de atributo*). *DadosConferencia.url* recebeu *Conferencia.url* (*mudança de atributo*). O objeto *DadosRegistroRevisao* foi criado (*criação de instância*) e associado ao *AgenteFormRevisao* (*formação de associação*). *DadosRegistroRevisao.id* recebeu *RegistroRevisao.id* (*mudança de atributo*). *DadosRegistroRevisao.qntCopias* recebeu *RegistroRevisao.qntCopias* (*mudança de atributo*). *DadosRegistroRevisao.dtEnvio* recebeu *RegistroRevisao.dtEnvio* (*mudança de atributo*). O objeto *DadosArtigo* foi criado (*criação de instância*) e associado a *DadosRegistroRevisao* (*formação de associação*). *DadosArtigo.id* recebeu *Artigo.id* (*mudança de atributo*). *DadosArtigo.titulo* recebeu *Artigo.titulo* (*mudança de atributo*). *DadosArtigo.arquivo* recebeu *Artigo.arquivo* (*mudança de atributo*). O objeto *DadosMembroComite* foi criado (*criação de instância*) e associado a *DadosRegistroRevisao* (*formação de associação*). *DadosMembroComite.nome* recebeu *MembroComite.nome* (*mudança de atributo*). O objeto *Destino* foi criado e associado a *DadosMembroComite* (*formação de associação*). *Destino.endRegiao* recebeu *MembroComite.endRegiao* (*mudança de atributo*). *Destino.nomeAgencia* recebeu *MembroComite.nomeAgencia* (*mudança de atributo*). O objeto *Itinerario* foi criado (*criação de instância*) e associado ao *AgenteFormRevisao* (*formação de associação*).

	O objeto *Destino* associado a *DadosMembroComite* foi associado ao *Itinerario* (*formação de associação*).

Contrato	
Nome:	**redirecionarAgenteFormRevisao(endRegiao, nomeAgencia, retornar)**
Responsabilidades:	Redirecionar o formulário de revisão para um revisor e requisitar ou não o seu retorno para o remetente.
Referências	Caso de uso: Revisar artigo, Redirecionar formulário de revisão.
Tipo:	Sistema
Exceções:	Se a agência do revisor estiver desconectada, o *AgenteFormRevisao* (agente móvel) deverá tentar migrar após um certo tempo. Se após um certo número de tentativas o *AgenteFormRevisao* não conseguir migrar, indicar erro.
Saída:	O *AgenteFormRevisao* informa que está migrando para a agência do revisor.
Pré-condições:	O estado do *AgenteFormRevisao* deve ser igual a EMREDIRECIONAMENTO ou EMREVISAO. A agência do revisor deve estar conectada.
Pós-condições:	Se *retornar* igual a *false,* o *Destino* atual foi removido do objeto *Itinerari*o (*quebra de associação*). Um novo objeto *Destino* foi criado (*criação de instância)* e associado ao objeto *Itinerario* (*formação de associação*). *Destino.endRegiao* recebeu o valor de *endRegiao* (*mudança de atributo*). *Destino.nomeAgencia* recebeu o valor de *nomeAgencia* (*mudança de atributo*). Se *AgenteFormRevisao.estado* = *EMDISTRIBUICAO, AgenteFormRevisao.estado* recebeu o valor *EMREVISAO*. *AgenteFormRevisao* (agente móvel) migrou para o próximo destino. (***migração***).

Contrato	
Nome:	**extrairArtigo(diretorio)**
Responsabilidades:	Extrair o arquivo que contém o artigo no diretório indicado pelo usuário.
Referências	Caso de uso: Revisar artigo, Redirecionar artigo
Tipo:	Sistema
Exceções:	Se o diretório não existir, indicar erro.
Saída:	Uma mensagem é exibida para o usuário informando que o arquivo foi extraído com sucesso.
Pré-condições:	O diretório deve existir na maquina do revisor.
Pós-Condições:	O *Artigo.arquivo* foi copiado para o diretório na máquina do revisor.

Contrato	
Nome:	**entrarDadosRevisao(nomeRevisor, instituicaoRevisor, endRevisor, foneRevisor, faxRevisor, emailRevisor, originalidade, legibilidade, relevancia, aceitacao, comentarios, comentariosSigilosos)**
Responsabilidades:	Entrar com as impressões do revisor sobre o artigo no que diz respeito à originalidade, legibilidade, relevância, aceitação e registrar os comentários abertos e sigilosos. Verificar se os parâmetros possuem valores válidos.
Referências	Caso de Uso: Revisar artigo, Aprovar revisão
Tipo:	Sistema
Exceções:	Se os parâmetros não possuírem valores válidos, indicar erro.
Saída:	
Pré-condições:	*AgenteFormRevisao.estado* deve possuir o valor EMREVISAO ou EMAPROVACAO. Os parâmetros passados devem possuir valores válidos.
Pós-condições:	Se o objeto *DadosRevisao* ainda não existia, ele foi criado (*criação de instância*) e associado ao *AgenteFormRevisao* (*formação de associação*). *DadosRevisao.originalidade* recebeu *originalidade* (*mudança de*

	atributo). *DadosRevisao.legibilidade* recebeu *legibilidade* (*mudança de atributo*). *DadosRevisao.relevancia* recebeu *relevancia* (*mudança de atributo*). *DadosRevisao.aceitacao* recebeu *aceitacao* (*mudança de atributo*). *DadosRevisao.comentarios* recebeu *comentarios* (*mudança de atributo*). *DadosRevisao.comentariosSigilosos* recebeu *comentariosSigilosos* (*mudança de atributo*). Se o objeto *DadosRevisor* ainda não existia, ele foi criado (*criação de instância)* e associado a *DadosRevisao* (*formação de associação*). *DadosRevisor.nome* recebeu *nomeRevisor* (*mudança de atributo*). *DadosRevisor.instituicao* recebeu *instituicaoRevisor* (*mudança de atributo*). *DadosRevisor.endereco* recebeu *enderecoRevisor* (*mudança de atributo*). *DadosRevisor.fone* recebeu fone (*mudança de atributo*). *DadosRevisor.fax* recebeu fax (*mudança de atributo*). *DadosRevisor.email* recebeu *emailRevisor* (*mudança de atributo*). *DadosRevisor.destino* recebeu o valor do destino atual (*mudança de atributo*).

Contrato	
Nome:	**finalizarRevisao( )**
Responsabilidades:	Registrar que o processo de revisão já foi completado e enviar o formulário de volta a agência do último remetente que solicitou o seu retorno. Caso ninguém o tenha solicitado, enviá-lo de volta ao coordenador de programa.
Referências	Caso de uso: Revisar artigo
Tipo:	Sistema

Exceções:	Se os dados de revisão não estiverem preenchidos, indicar erro Se a agência do receptor estiver desconectada, o *AgenteFormRevisao* (agente móvel) deverá tentar migrar após um certo tempo. Se após um certo número de tentativas o *AgenteFormRevisao* não conseguir migrar, indicar erro.
Saída:	O *AgenteFormRevisao* informa que está retornando para a agência do coordenador.
Pré-condições:	*AgenteFormRevisao.estado* deve possuir o valor EMREVISAO. Os dados revisão do formulário devem estar preenchidos. A agência do receptor deve estar conectada.
Pós-condições:	*AgenteFormRevisao.estado* recebeu o valor *EMAPROVACAO* (*mudança de atributo*). *Destino* atual foi removido do objeto *Itinerari*o (*quebra de associação*). *AgenteFormRevisao* (agente móvel) retornou para agência do último revisor que solicitou o seu retorno (***migração****).* Se o membro do comitê ou nenhum revisor solicitou o retorno do *AgenteFormRevisao,* este retornou para agência do coordenador (***migração****).* O objeto *FormRevisao* foi criado (*criação de instância*) e associado a *RegistroRevisao* (*formação de associação*) *FormRevisao.originalidade* recebeu *DadosRevisao.originalidade* (*mudança de atributo*). *FormRevisao.legibilidade* recebeu *DadosRevisao.legibilidade* (*mudança de atributo*). *FormRevisao.relevancia* recebeu *DadosRevisao.relevancia* (*mudança de atributo*). *FormRevisao.aceitacao* recebeu *DadosRevisao.aceitacao* (*mudança de atributo*). *FormRevisao.comentarios* recebeu *DadosRevisao.comentarios* (*mudança de atributo*). *FormRevisao.comentariosSigilosos* recebeu

	DadosRevisao.comentariosSigilosos (*mudança de atributo*). O objeto *Revisor* foi criado (*criação de instância)* e associado a *FormRevisao (formação de associação).* *Revisor.nome* recebeu *DadosRevisor.nome* (*mudança de atributo*). *Revisor.instituicao* recebeu *DadosRevisor.instituicao* (*mudança de atributo*). *Revisor.endereco* recebeu *DadosRevisor.endereco* (*mudança de atributo*). *Revisor.fone* recebeu *DadosRevisor.fone* (*mudança de atributo*). *Revisor.fax* recebeu *DadosRevisor.fax* (*mudança de atributo*). *Revisor.email* recebeu *DadosRevisor.email* (*mudança de atributo*). *Revisor.endRegiao* recebeu *DadosRevisor.destino.endRegiao* (*mudança de atributo*). *Revisor.nomeAgencia* recebeu *DadosRevisor.destino.nomeAgencia* (*mudança de atributo*). *AgenteFormRevisao* foi destruído (*destruição de instância*).

Contrato	
Nome:	**desativarAgenteFormRevisao ( )**
Responsabilidades:	Desativar o *AgenteFormRevisao*.
Referências	Casos de uso: Redirecionar revisão, Revisar artigo, Aprovar revisão
Tipo:	Sistema
Exceções:	Se o *AgenteFormRevisao* já estiver desativado, indicar erro.
Saída:	O AgenteFormRevisao informa que está sendo.desativado.
Pré-condições:	O *AgenteFormRevisao* deve estar ativo.
Pós-condições:	*AgenteFormRevisao* foi desativado e persistido (*desativação).*

Contrato	
Nome:	**ativarAgenteFormRevisao( )**
Responsabilidades:	Ativar o AgenteFormRevisao.

Referências	Casos de uso: Redirecionar revisão, Revisar artigo, Aprovar revisão.
Tipo:	Sistema
Exceções:	Se o *AgenteFormRevisao* já estiver ativado, indicar erro.
Saída:	Exibir formulário de revisão.
Pré-condições:	O *AgenteFormRevisao* deve estar desativado.
Pós-condições:	*AgenteFormRevisao* foi ativado (*ativação).*

Contrato	
Nome:	**aprovarRevisao( )**
Responsabilidades:	Encaminhar o AgenteFormRevisao de volta a agência do último remetente que solicitou o seu retorno. Caso ninguém o tenha solicitado, enviá-lo de volta ao coordenador de programa.
Referências	Caso de uso: Aprovar revisão
Tipo:	Sistema
Exceções:	*AgenteFormRevisao.estado* deve possuir o valor EMAPROVACAO. Os dados revisão do formulário devem estar preenchidos. A agência do receptor deve estar conectada. Se a agência do receptor estiver desconectada, o *AgenteFormRevisao* (agente móvel) deverá tentar migrar a após um certo tempo. Se após um certo número de tentativas o *AgenteFormRevisao* não conseguir migrar, indicar erro.
Saída:	O AgenteFormRevisao informa que está retornando para a agência do coordenador.
Pré-condições:	*AgenteFormRevisao* deve existir. Os atributos do *FormRevisao*: *legibilidade, relevancia* e *aceitação*, devem estar preenchidos. Todos os atributos devem possuir valores válidos. A agência do receptor deve estar conectada.
Pós-condições:	*Destino* atual foi removido do objeto *Itinerari*o (*quebra de associação*). *AgenteFormRevisao* (agente móvel) retornou para agência do

	último revisor que solicitou o seu retorno (***migração***). Se o membro do comitê ou nenhum revisor solicitou o retorno do *AgenteFormRevisao,* este retornou para agencia do coordenador (***migração***). O objeto *FormRevisao* foi criado (*criação de instância*) e associado a *RegistroRevisao* (*formação de associação*) *FormRevisao.originalidade* recebeu *DadosRevisao.originalidade* (*mudança de atributo*). *FormRevisao.legibilidade* recebeu *DadosRevisao.legibilidade* (*mudança de atributo*). *FormRevisao.relevancia* recebeu *DadosRevisao.relevancia* (*mudança de atributo*). *FormRevisao.aceitacao* recebeu *DadosRevisao.aceitacao* (*mudança de atributo*). *FormRevisao.comentarios* recebeu *DadosRevisao.comentarios* (*mudança de atributo*). *FormRevisao.comentariosSigilosos* recebeu *DadosRevisao.comentariosSigilosos* (*mudança de atributo*). O objeto *Revisor* foi criado (*criação de instância)* e associado a *FormRevisao (formação de associação).* *Revisor.nome* recebeu *DadosRevisor.nome* (*mudança de atributo*). *Revisor.instituicao* recebeu *DadosRevisor.instituicao* (*mudança de atributo*). *Revisor.endereco* recebeu *DadosRevisor.endereco* (*mudança de atributo*). *Revisor.fone* recebeu *DadosRevisor.fone* (*mudança de atributo*). *Revisor.fax* recebeu *DadosRevisor.fax* (*mudança de atributo*). *Revisor.email* recebeu *DadosRevisor.email* (*mudança de atributo*). *Revisor.endRegiao* recebeu *DadosRevisor.destino.endRegiao* (*mudança de atributo*). *Revisor.nomeAgencia* recebeu

	DadosRevisor.destino.nomeAgencia (*mudança de atributo*). *AgenteFormRevisao* foi destruído (*destruição de instância*).

4.5 Validação

A fim de validar os artefatos produzidos durante esta fase as seguintes atividades devem ser realizadas:

- Verificar a completude, a corretude e a consistência dos diagramas de seqüência do sistema e dos diagramas de atividades com relação aos casos de uso.
- Verificar a completude, a corretude e a consistência dos contratos das operações do sistema com relação aos casos de uso e aos diagramas de seqüência do sistema.
- Discutir e validar com o cliente todos os artefatos gerados.

4.6 Considerações Finais

Este capítulo apresentou a fase de análise do modelo proposto para o desenvolvimento de aplicações baseadas em agentes móveis. O principal objetivo desta fase é gerar um modelo de alto nível do domínio do problema que deve ser voltado para o entendimento do usuário, mas que também será utilizado pelos projetistas do sistema servindo de base para a execução das próximas fases.

Os artefatos produzidos nesta fase são bem semelhantes aos gerados na fase de análise convencional, com a exceção de que, em nosso modelo, conceitos de agentes são introduzidos gradualmente a partir das especificações dos casos de usos. Desde que os conceitos de agentes levantados sejam intrínsecos ao domínio do problema, eles devem ser mencionados na fase de análise, mas apenas como conceitos e não como entidades de software. Desta forma, conceitos de agentes e objetos passam a figurar lado a lado no modelo conceitual. Diagramas de seqüência do sistema, diagramas de atividades e contratos de operações do sistema são produzidos para investigar o comportamento do sistema como uma caixa preta, ou seja, o que ele faz e não como ele faz. Nos diagramas de sequência do sistema e nos diagramas de ativades do modelo proposto agentes são ilustrados como parte do sistema. Os contratos de operações de sistema descrevem o que cada operação promete cumprir através das pós-condições, ou seja, o que muda no estado do sistema após a execução da operação. Em nosso modelo, além das categorias convencionais de pós-condições (criação e destruição de instâncias, modificação de atributos, associações formadas e quebradas) também poderão ser identificadas as seguintes outras em um sistema baseado em agentes móveis: migração,

quando o agente migra para outra localização; desativação, quando o agente é desativado; e ativação, quando o agente é ativado.

Para auxiliar nas descrições dos artefatos, exemplos gerados durante a fase de análise do estudo de caso Sistema de Apoio a Comitês de Programa em Conferências foram ilustrados. É importante ressaltar que nos baseamos na especificação dada por Cardelli em nosso estudo de caso e que por este motivo nos pareceu natural introduzir o conceito de agentes a partir das especificações de casos de uso. Em outras especificações do mesmo sistema dificilmente o analista sentiria esta mesma naturalidade.

A partir do próximo capítulo estaremos falando da fase de projeto, onde os artefatos são voltados para o entendimento do projetista. Esta fase é dividida em: projeto arquitetural, projeto detalhado independente de plataforma e projeto detalhado dependente de plataforma. O Capítulo 5 descreverá a fase de projeto arquitetural.

5 PROJETO ARQUITETURAL

Este capítulo apresenta a fase de projeto arquitetural do modelo proposto que corresponde ao nível de projeto de macroarquitetura apresentado em [TOH99].

Em nosso modelo, o projeto arquitetural representa o esboço das configurações do sistema, bem como o comportamento dos agentes, que é definido através da aplicação dos padrões de mobilidade propostos por Tahara et al [TOH99] para o nível de macroarquitetura.

A seguir, o objetivo da fase de projeto arquitetural, bem como seus artefatos de entrada, as tarefas a serem realizadas, os critérios de Validação e os artefatos a serem produzidos serão descritos.

5.1 Objetivo

Definir a arquitetura do sistema e representar o comportamento dos agentes em alto nível.

5.2 Entradas

Durante a elaboração da fase de projeto arquitetural devemos ter em mãos o documento de especificações de requisitos, visto que a arquitetura do sistema deve garantir que os requisitos funcionais e não funcionais do sistema possam ser cumpridos.

5.3 Tarefas

As principais tarefas a serem realizadas nesta fase consistem em:

- Descrever a arquitetura do sistema
    - Decompor o sistema em camadas e partições
    - Elaborar um diagrama ilustrando a decomposição do sistema em camadas e partições
    - Decidir a localização do *code base* dos agentes móveis
    - Esboçar a configuração do sistema ilustrando a localização física das camadas
- Elaborar diagramas de comportamento dos agentes para os cenários mais importantes do sistema, aplicando os padrões de mobilidade propostos por Tahara et al [TOH99] para o nível de projeto de macroarquitetura, que é correspondente ao projeto arquitetural do nosso modelo

- Escolher a plataforma de agentes móveis

5.4 Artefatos

5.4.1 Projeto Arquitetural

O projeto arquitetural é gerado durante a fase de Projeto. Ao criá-lo devemos levar em consideração os requisitos funcionais e não funcionais do sistema, a fim de garantir que eles possam ser cumpridos.

Ao elaborarmos um projeto arquitetural decisões estratégicas de alto nível terão que ser tomadas, tais como:

- Modularização do projeto em subsistemas
- Escolha de uma estrutura de comunicação e controle entre os subsistemas
- Definição das interfaces entre subsistemas
- Decisão sobre uso/criação de bibliotecas e/ou componentes
- Escolha de uma forma básica de persistência (arquivo, memória, SGBD)
- Escolha do paradigma de SGBD a usar (relacional, objeto relacional, orientado a objeto)
- Escolha de uma estratégia de interação entre a aplicação e os dados
- Atendimento a requisitos especiais de desempenho
- Atendimento a outros requisitos (custo, mobilidade, uso de padrões, etc.)

Quando o projeto arquitetural está sendo elaborado para um sistema baseado em agentes móveis surgem algumas decisões adicionais:

- Definir o comportamento dos agentes, através da aplicação dos padrões de mobilidade propostos por Tahara et al [TOH99].
- Decidir o tipo de *code base* (lugar onde as classes dos agentes podem ser acessadas pelas agências quando estas precisam criar um agente ou reinstanciá-lo depois de sua migração). Por exemplo, a plataforma Grasshopper suporta dois tipos de *code base*: sistema de arquivo e servidor HTTP. No primeiro tipo as classes dos agentes são mantidas pelo sistema de arquivo das agências e no segundo são mantidas em um servido HTTP.
- Escolher a plataforma de agentes móveis

Não existe uma estrutura rígida para um projeto arquitetural de um sistema baseado em agentes móveis, mas nós sugerimos que ele seja estruturado como segue:

5.4.1.1 Descrição da Arquitetura do Sistema

Esta seção tem o objetivo de ilustrar a arquitetura do sistema, esboçar a sua configuração física e a localização do *code base* dos agentes. A seguir apresentaremos uma sugestão para a arquitetura do Sistema de Apoio às Atividades de Comitês de Programa em Conferências.

Na Figura 5.1 temos a decomposição em camadas do sistema:

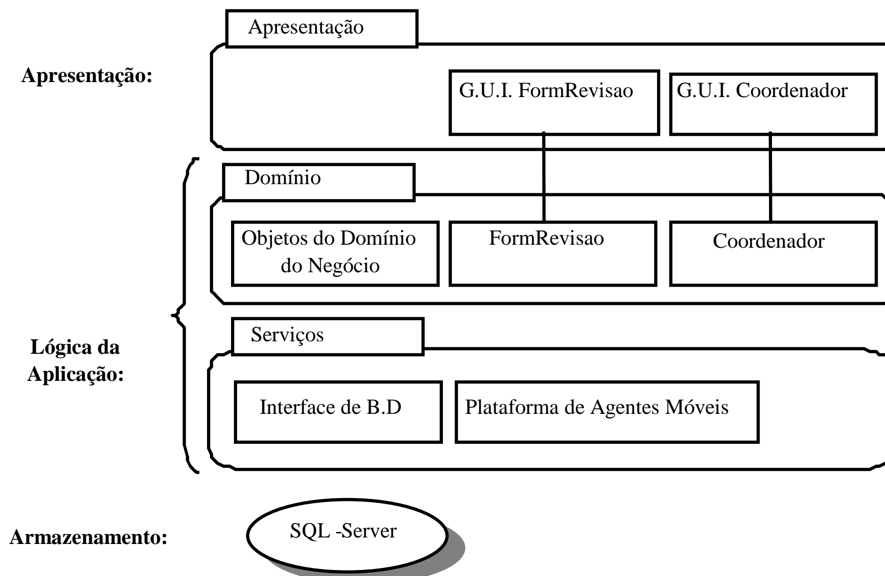

Figura 5.1: Decomposição em camadas do sistema

Note que o sistema é decomposto em três camadas verticais:

- Camada de apresentação: suportada pelos próprios agentes através de suas interfaces gráficas. Será através da interface gráfica dos agentes que os usuários (coordenador do programa, membros de comitê e revisores) irão interagir com o sistema.
- Camada lógica da aplicação: quebrada em duas outras camadas, uma relativa aos aspectos do domínio da aplicação e outra aos aspectos de serviço.
- Camada de armazenamento: consiste no banco de dados SQL Server, que utiliza o paradigma relacional

A configuração física do sistema e a localização do *code base* dos agentes é ilustrada na Figura 5.2:

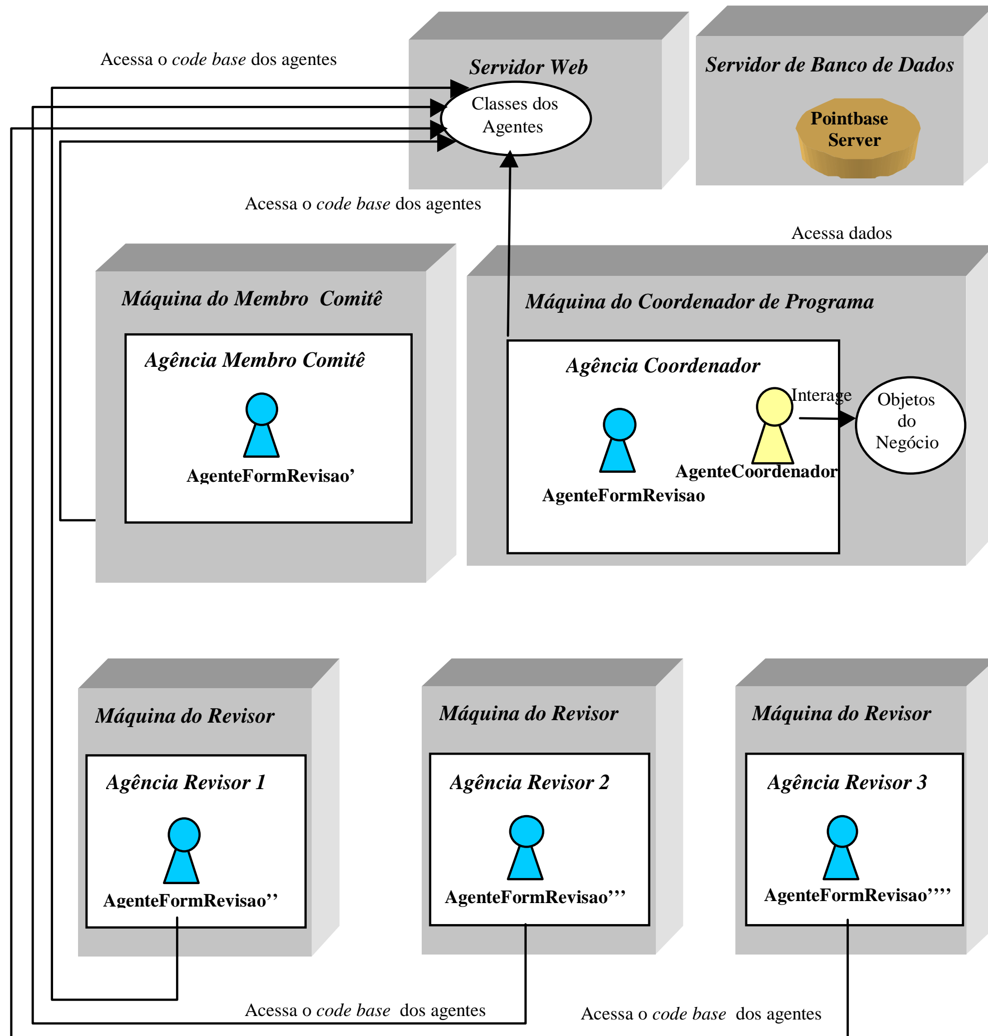

Figura 5.2: Configuração física do sistema

Todos os elementos da rede que receberão e rodarão agentes possuem uma agência, formando um ambiente de agentes distribuídos.

Os códigos das classes dos agentes irão residir em um Servidor Web. Sendo assim, para que uma agência possa criar ou reinstanciar um agente móvel, ela precisa solicitar o código da classe ao servidor. Esta decisão foi tomada para evitar a replicação do código em todas as máquinas.

Nesta iteração consideraremos que a máquina do coordenador é um dispositivo estático e está conectado através de rede local com o Servidor de Banco de Dados e com o Servidor Web. Já as máquinas dos revisores poderão consistir de dispositivos móveis, tais como *notebooks* ou *palmtops* e poderão estar localizados fisicamente em domínios diferentes.

5.4.1.2 Descrição do comportamento dos agentes

Esta seção tem o objetivo de representar em alto-nível o comportamento de mobilidade dos agentes. Para tal, os padrões de mobilidade propostos por Tahara et al [TOH99] para o nível de projeto de macroarquitetura devem ser aplicados e diagramas de comportamento dos agentes para os cenários mais importantes do sistema devem ser gerados.

5.4.1.2.1 Padrões de Mobilidade

Estes padrões são aplicados nas situações em que os agentes executam tarefas enquanto migram pela rede. O uso destes padrões pode melhorar a eficiência das redes, visto que os recursos são usados localmente e as redes são usadas apenas para transferência dos agentes e dos dados necessários. É importante observar que a quantidade de dados a ser transferida através da rede deve ser pequena para que estes padrões sejam eficazes. Os seguintes padrões de mobilidade de agentes foram identificados por Tahara et al [TOH99]: itinerário, movimento estrela (*star-shaped)* e ramificação *(branching).*

No padrão itinerário (Figura 5.3), os agentes migram de uma máquina destino para outra, realizando tarefas em cada uma delas. Este padrão é recomendado quando o usuário quer usar os recursos de uma rede com largura de banda relativamente pequena e as máquinas que possuem os recursos necessários são estáveis e conhecidas antecipadamente. Entretanto, se a quantidade de dados que os agentes carregam durante o itinerário for grande, este padrão torna-se inadequado devido ao aumento do tráfego da rede.

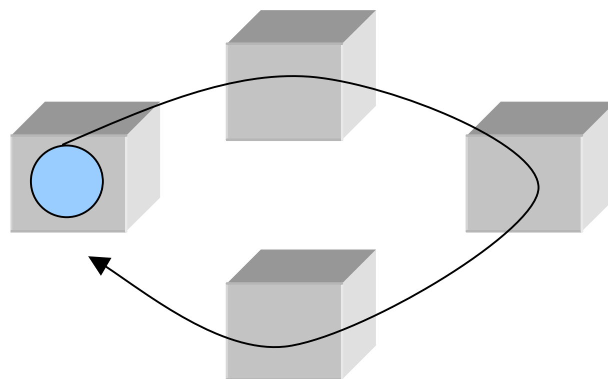

Figura 5.3: Padrão itinerário

No padrão movimento estrela (*star-shaped)*, os agentes migram da máquina origem para uma máquina destino onde realizarão tarefas e retornarão logo em seguida para a máquina origem, antes de migrar para outra máquina destino (Figura 5.4). O agente repete este

movimento até completar a sua tarefa. Este padrão deve ser aplicado no lugar do Itinerário, quando a quantidade de dados a ser transportada pelos agentes for considerável.

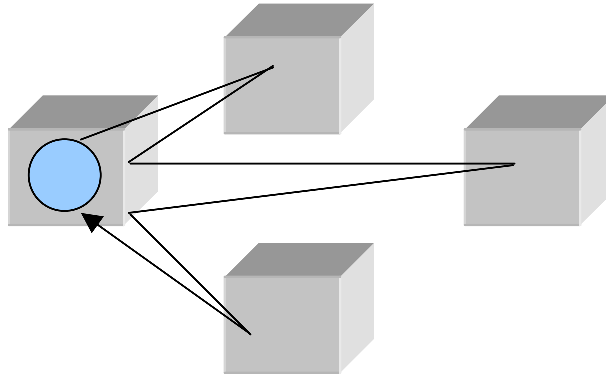

Figura 5.4: Padrão movimento estrela

No padrão ramificação (*branching*), uma cópia do agente original é enviada para cada máquina destino (Figura 5.5). Depois que cada cópia migra para as máquinas destinos, elas acessam os recursos destas e retornam para a máquina de origem, onde permanecem ou se destroem. Para aproveitar os resultados trazidos pelos agentes, podemos utilizar o padrão Fusão (*Merger)*. Este padrão possui duas variações: fusão síncrona – os resultados são unificados; fusão de seleção – o resultado de um só agente é considerado e os demais são abandonados. Este padrão tem a vantagem de aliviar o tráfego da rede, sendo apropriado para aplicações onde as máquinas são instáveis e o volume de dados a ser comunicado através da rede é grande.

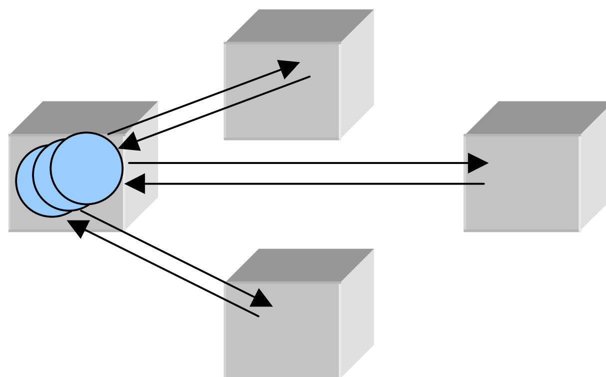

Figura 5.5: Padrão ramificação

5.4.1.2.2 Diagrama de comportamento dos agentes

Um diagrama de comportamento dos agentes descreve a configuração física dos sistemas distribuídos tais como as máquinas e as configurações de rede, como também a

forma que os agentes se comportam nestas configurações. Diagramas de comportamento dos agentes devem ser gerados para os cenários mais importantes do sistema.

No nosso estudo de caso, para representar o comportamento do agente móvel *AgenteFormRevisao*, aplicamos uma combinação dos padrões de mobilidade itinerário e ramificação, onde uma cópia do agente original é enviada para cada máquina destino e estas cópias podem migrar para outras máquinas destinos, através de um itinerário e realizar tarefas em cada uma delas. A Figura 5.6 ilustra o comportamento de mobilidade do *AgenteFormRevisao* segundo a combinação destes padrões.

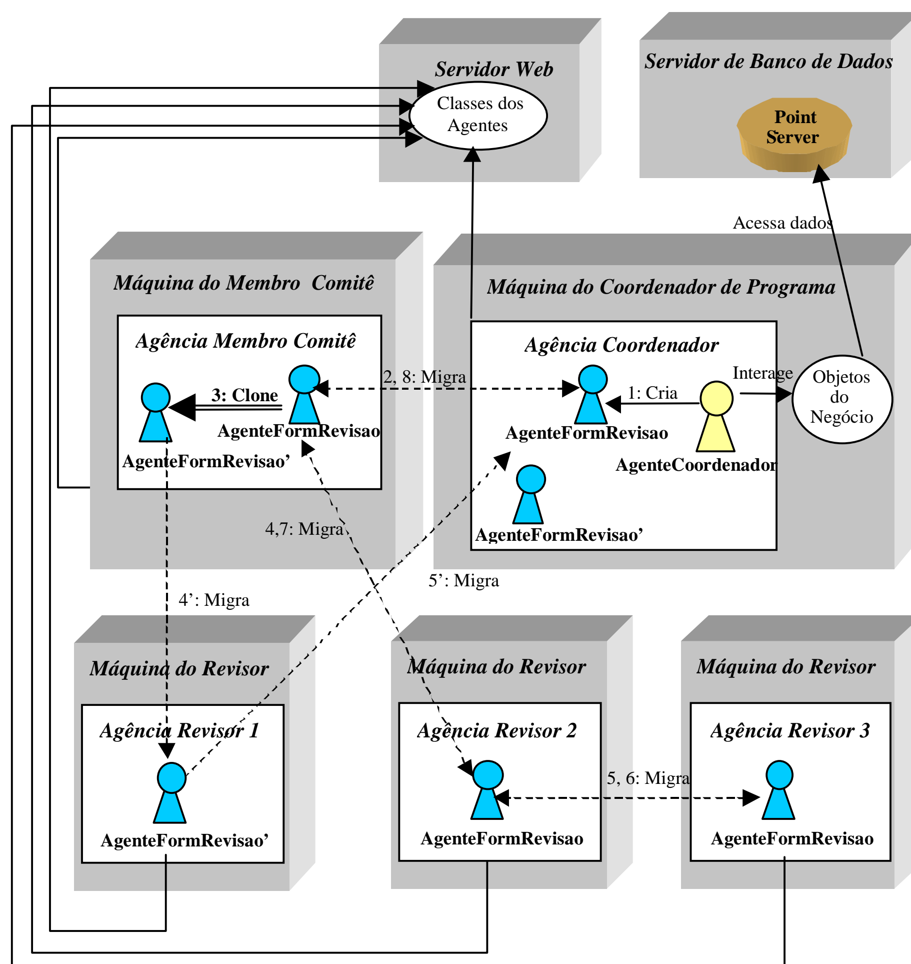

Figura 5.6: Diagrama de comportamento

Descrição do cenário. Através de uma interface gráfica, o coordenador do programa interage com o agente estacionário *AgenteCoordenador* dando início ao processo de revisão dos artigos. O coordenador seleciona um artigo, o membro do comitê responsável por encontrar revisores para este artigo e a quantidade N de cópias deste formulário que o membro do comitê deverá redirecionar. O *AgenteCoordenador* cria um *AgenteFormRevisao*

(agente móvel), já com o artigo incorporado (1). O *AgenteFormRevisao* migra para a máquina do membro do comitê (2). *AgenteFormRevisao* gera N-1 clones (3). O membro do comitê redireciona os agentes *AgenteFormRevisao* existentes em sua máquina para outros revisores, informando se deseja que eles retornem quando tiverem sido revisados. Os agentes *AgenteFormRevisao* migram para as máquinas dos revisores (4, 4'). O revisor recebe o *AgenteFormRevisao* e pode optar por revisar o artigo diretamente, ou redirecioná-lo para outro revisor, informando se deseja que eles retornem quando tiverem sido revisados (5). Quando o processo de revisão é finalizado o *AgenteFormRevisao* retorna direto para a máquina do coordenador de programa (5') caso nenhum dos remetentes tenha solicitado o seu retorno. Do contrário, o *AgenteFormRevisao* volta para o último remetente que solicitou o seu retorno (6). Caso ainda exista um remetente anterior a este, que também tenha solicitado o seu retorno, o Formulário de Revisão retornará para ele (7). Após ter sido aprovado por todos os remetentes que solicitaram o seu retorno, o *AgenteFormRevisao* volta para a máquina do coordenador de programa (8).

A cooperação existente entre o agente estacionário *AgenteCoordenador* e o agente móvel *AgenteFormRevisao* segue o padrão de Interação Direta, ou seja, os agentes se comunicam diretamente através da troca de mensagens. De acordo com este padrão, esta troca de mensagens geralmente ocorre através da rede, mas no nosso estudo de caso ela é realizada localmente na máquina do coordenador de programa.

5.4.1.3 Plataforma de Agentes Móveis Escolhida

Para o desenvolvimento do nosso estudo de caso escolhemos a plataforma de agentes móveis Grasshopper. Dentre as plataformas baseadas em Java (Aglets, Voyager, Concordia) que foram estudadas esta foi a escolhida dentre outros motivos por ser a primeira plataforma de agentes móveis que segue o padrão OMG-MASIF, por possibilitar a integração do paradigma cliente-servidor com tecnologia de agentes móveis e por estar disponível gratuitamente.

5.5 Validação

Verificar a completude do projeto arquitetural com relação aos requisitos funcionais e não funcionais.

5.6 Considerações Finais

Este capítulo apresentou a fase de projeto arquitetural do modelo proposto para o desenvolvimento de aplicações baseadas em agentes móveis. O principal objetivo desta fase é gerar a arquitetura do sistema e representar o comportamento dos agentes em alto nível.

Na fase de projeto arquitetural do nosso modelo, além das decisões de alto nível tomadas para sistemas convencionais, outras específicas para sistemas baseados em agentes móveis precisam ser consideradas, entre elas precisamos definir o comportamento dos agentes, a localização das suas classes e a plataforma de agentes móveis onde aplicação será desenvolvida.

Afim ilustrar o comportamento dos agentes, diagramas para os cenários mais importantes do sistema devem ser gerados através da aplicação dos padrões de mobilidade apresentados em [TOH99]. Os padrões de mobilidade utilizados no projeto arquitetural influenciarão a escolha de padrões de projeto para agentes móveis na fase de projeto detalhado independente de plataforma, onde decisões de baixo-nível serão tomadas a fim de produzir a especificação lógica do sistema. A fase de projeto detalhado independente de plataforma será apresentada no Capítulo 6.

6 PROJETO DETALHADO INDEPENDENTE DE PLATAFORMA

Este capítulo apresenta a fase de projeto detalhado independente de plataforma que trata de todo o projeto lógico da aplicação sem entrar em detalhes relacionados a uma plataforma de agentes móveis específica. Os principais artefatos gerados nesta fase são os diagramas de classe e os diagramas de interação que, através da utilização da extensão de UML proposta por Klein et al [KRSW01], são capazes de representar regiões, agências, agentes móveis, como também mobilidade e clonagem de agentes.

A seguir, o objetivo da fase de projeto detalhado independente de plataforma, bem como seus artefatos de entrada, as tarefas a serem realizadas, os artefatos a serem produzidos e os critérios de validação serão descritos.

6.1 Objetivo

Produzir uma especificação lógica independente de qualquer plataforma de agentes móveis. Os artefatos gerados nesta fase devem refletir como o comportamento do agente é implementado para cumprir os requisitos funcionais e não funcionais do sistema. Nesta fase, padrões de agentes móveis independentes de plataforma devem ser aplicados.

6.2 Entradas

Os artefatos gerados na fase de projeto independente de plataforma dependem dos artefatos produzidos nas fases de análise e de projeto arquitetural.

6.3 Tarefas

As principais tarefas a serem executadas nesta fase consistem em criar diagramas de interação para cada operação do sistema, gerar diagramas de classes e, no caso de aplicações que necessitam persistir suas informações, gerar o esquema do banco de dados.

6.4 Artefatos

6.4.1 Diagramas de Interação

Os diagramas de interação têm o objetivo de mostrar como as pós-condições dos contratos serão realizadas e são elaborados durante a fase de projeto detalhado. Sua criação é dependente dos seguintes artefatos:

- Modelo Conceitual: é a partir dele que o projetista define as classes de software correspondentes aos conceitos. Objetos destas classes participam das interações ilustradas nos diagramas de interação.
- Contratos das operações do sistema: é através deles que o projetista identifica as responsabilidades e pós-condições que os diagramas de interação devem cumprir.
- Diagrama de Comportamento dos Agentes: é a partir dele que o projetista identifica o padrão de mobilidade dos agentes que deverá ser adotado.

Dois tipos de diagramas de interação podem ser usados para mostrar como os objetos e os agentes interagem (através de mensagens) para realizar tarefas:

- Diagramas de Seqüência
- Diagramas de Colaboração

O diagrama de seqüência é mais simples de usar quando se deseja mostrar apenas as seqüências de interações. Enquanto que o diagrama de colaboração é mais adequado quando se deseja expressar mais detalhes da colaboração entre objetos, tal como o tipo de visibilidade entre eles.

A fim de representar aspectos de agentes móveis nos diagramas de interação utilizaremos as seguintes extensões de UML propostas por Klein et al [KRSW01]:

- O estereótipo *<<region>>*, derivado do metamodelo *package*, especifica um mecanismo para organização de agências dentro de grupos. Graficamente, uma região é representada como na Figura 6.1.

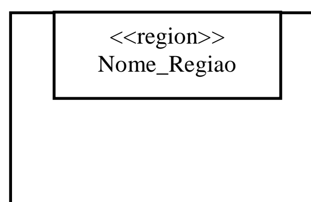

Figura 6.1: Representação gráfica de uma região em um diagrama de interação

- O estereótipo <<*agency*>>, também derivado do metamodelo *package*, foi introduzido para expressar mais claramente o relacionamento entre uma agência e seus agentes. Uma agência agrupa o conjunto de agentes que estão hospedados nela naquele momento. Graficamente, uma agência é representada como na Figura 6.2.

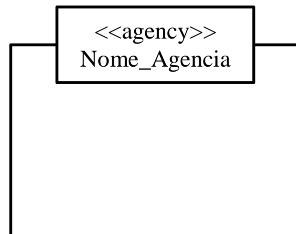

Figura 6.2: Representação gráfica de uma agência em um diagrama de interação

- O estereótipo <<*mobile agent*>>, derivado do metamodelo *class*, foi introduzido para diferir um agente móvel de um objeto comum. Graficamente, um agente móvel é representado como na Figura 6.3.

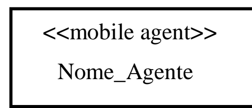

Figura 6.3: Representação gráfica de um agente móvel

- O estereótipo <<move>>, derivado do metamodelo *dependency*, especifica uma dependência entre um agente fonte e o agente alvo. A dependência especifica que o objeto agente fonte e o objeto agente alvo representam a mesma instância em momentos e em agências diferentes com o mesmo estado de execução, isto é, o agente continua sua execução exatamente do ponto em que parou antes da migração. Graficamente, uma migração é representada como na Figura 6.4.

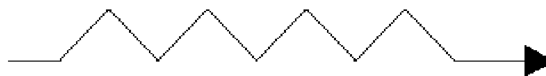

Figura 6.4: Representação gráfica de uma migração

- O estereótipo <<clone>>, derivado do metamodelo *dependency*, especifica uma dependência entre um agente fonte e o agente alvo. A dependência especifica que o objeto agente fonte cria o objeto agente alvo como uma cópia exata de si próprio. O objeto alvo possui a mesma informação interna, isto é o mesmo estado de execução, e assim ele começa a executar exatamente no ponto em o objeto agente fonte havia

alcançado quando o objeto agente alvo foi criado. Graficamente, uma clonagem é representada como na Figura 6.5.

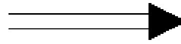

Figura 6.5: Representação gráfica de uma clonagem

Nós identificamos ainda a necessidade de mais uma extensão:

- O estereótipo <<*stationary agent*>>, derivado do metamodelo *class*, foi introduzido para representar um agente estacionário. Graficamente, um agente estacionário é representado como na Figura 6.6.

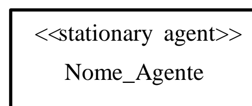

Figura 6.6: Representação gráfica de um agente estacionário

Ao elaborar diagramas de interação as seguintes recomendações devem ser seguidas:

- Criar um diagrama separado para cada operação do sistema sendo desenvolvida na iteração corrente. Para cada mensagem de operação do sistema, um diagrama é construído com essa mensagem inicial.
- Se o diagrama ficar complexo (não cabe numa única página), quebre-o em diagramas menores.
- Usando o contrato das operações (principalmente as pós-condições) e os casos de uso como ponto de partida, projete um sistema de objetos e agentes interagindo entre si para realizar tarefas. Aplique padrões GRASP (*General Responsability Assignment Software Patterns*) [Lar02], padrões de projeto O.O. [GHJV95] e padrões de agentes móveis [LO98] para realizar um bom projeto.

Em nosso estudo de caso uma adaptação do padrão Mestre-Escravo Intinerante (*Master-Slave Revisited)* proposto por Lange et al em [LO98], que consiste na combinação dos padrões Mestre-Escravo e Itinerário, está sendo aplicada.

O padrão Mestre-Escravo define um esquema onde um agente mestre pode delegar tarefas a um agente escravo. O agente escravo migra para uma agência remota a fim de realizar a sua tarefa e retorna o resultado desta para o agente mestre.

O padrão Itinerário define um objeto associado ao agente móvel que detém o conhecimento do itinerário do agente. É este objeto que define qual será o próximo destino do agente.

Na combinação destes dois padrões, o agente escravo migra por várias agências remotas de acordo com o seu itinerário. Em cada uma dessas agências, o agente escravo realiza a sua tarefa e retorna o resultado dela para o agente mestre.

Em nosso estudo de caso, o agente mestre é representado pelo *AgenteCoordenador* e os agentes escravos pelos agentes da classe *AgenteFormRevisao.* O objeto *Itinerario* está sendo utilizado com o objetivo de guardar o caminho de volta do *AgenteFormRevisao*. Esta foi a solução adotada para cumprir o requisito de sistema que estabelece que um revisor pode solicitar o retorno do *AgenteFormRevisao,* com o objetivo de checar a revisão dos subrevisores antes que o agente retorne para o coordenador de programa. Desta forma, os destinos do itinerário vão sendo adicionados a ele, inicialmente durante a sua criação quando ele recebe o seu primeiro destino (agência do membro de comitê) e depois à medida que o *AgenteFormRevisao* é redirecionado. Sempre que o *AgenteFormRevisao* é redirecionado, o revisor tem a opção de solicitar o seu retorno. Caso ele não o faça, o endereço da agência deste revisor será removido do itinerário. Quando o processo de revisão é concluído o *AgenteFormRevisao* faz o caminho de volta para a agência do coordenador de programa, parando nas agências dos revisores ou do membro do comitê que solicitaram o seu retorno. Ao chegar na agência do coordenador de programa, o *AgenteFormRevisao* entrega os dados da revisão para o *AgenteCoordenador* e se destrói. As classes, interfaces e associações geradas devido ao uso deste padrão estão ilustradas na Figura 6.22, enquanto que o comportamento descrito acima está ilustrado nos diagramas de seqüência e de colaboração das operações de sistema *gerarFormRevisao, redirecionarAgenteFormRevisao, finalizarRevisao* e *aprovarRevisao.*

6.4.1.1 Diagramas de interação da operação de sistema *gerarFormRevisao*

Na Figura 6.7, primeira parte do diagrama de seqüência da operação de sistema *gerarFormRevisao*, o ator coordenador de programa inicia o processo de revisão de um artigo, solicitando ao *AgenteCoordenador* que um formulário de revisão seja criado para o artigo e que *N* cópias deste sejam entregues a um dado membro de comitê. Para isto, o *AgenteCoordenador* interage com o objeto externo *Conferencia,* para coletar informações sobre a conferência, e com a sua agência, para solicitar a criação do *AgenteFormRevisao.* A

Figura 6.8 corresponde ao diagrama de colaboração da primeira parte do diagrama de sequência da operação de sistema *gerarFormRevisao*.

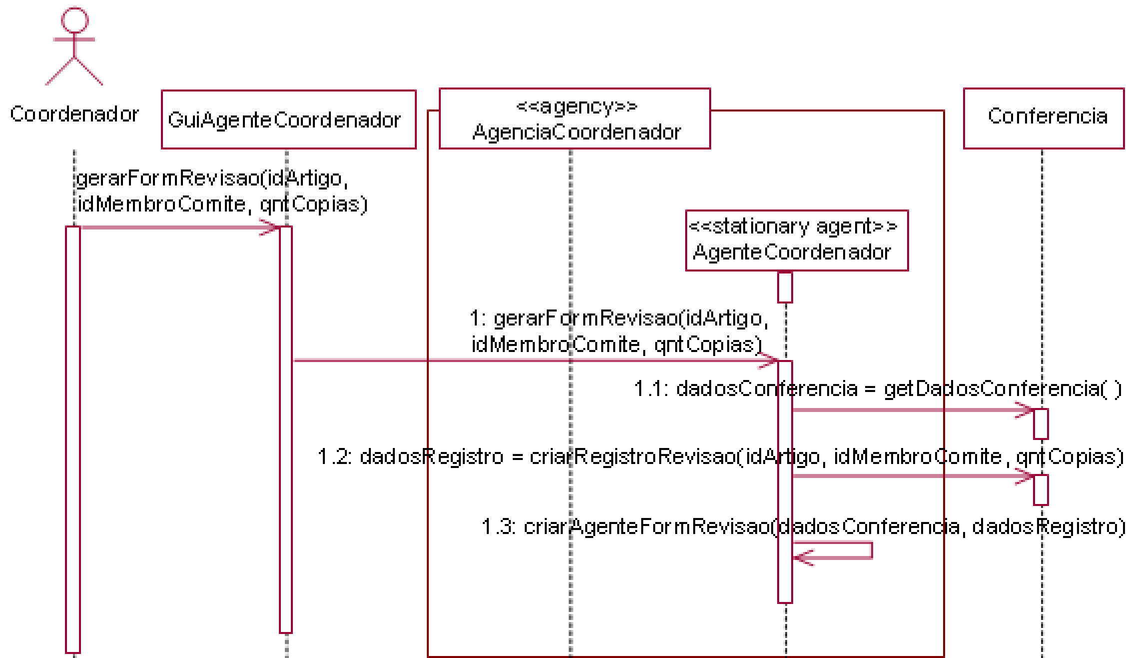

Figura 6.7: Diagrama de sequência da operação de sistema *gerarFormRevisao* – Parte 1

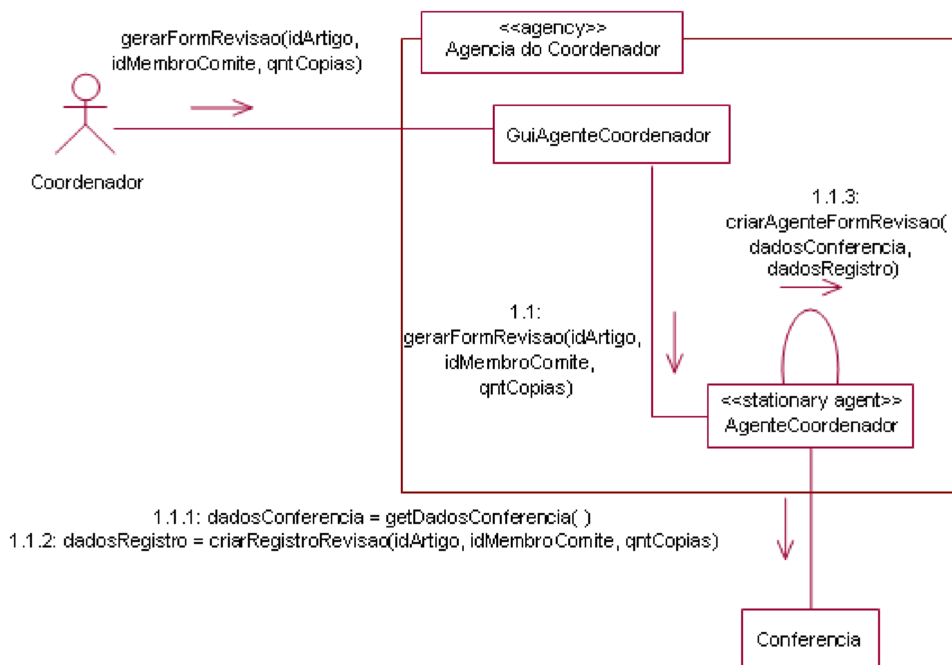

Figura 6.8: Diagrama de colaboração da operação de sistema *gerarFormRevisao* – Parte 1

Na Figura 6.9, que dá continuação ao diagrama anterior, o *AgenteCoordenador* solicita à agência na qual ele está hospedado que crie o *AgenteFormRevisao.* Logo após o momento da sua criação o *AgenteFormRevisao* inicia a sua tarefa, inicializando o valor do atributo estado para *CRIADOAGORA* e criando o objeto *Itinerário* que receberá o seu primeiro destino (endereço da agência do membro comitê) e uma referência para o *AgenteFormRevisao*. Em seguida o *AgenteFormRevisao,* altera o valor do atributo estado para EMCLONAGEM e chama o método *migra* para se transferir para o próximo destino do seu itinerário.. A Figura 6.10 ilustra o diagrama de colaboração correspondente

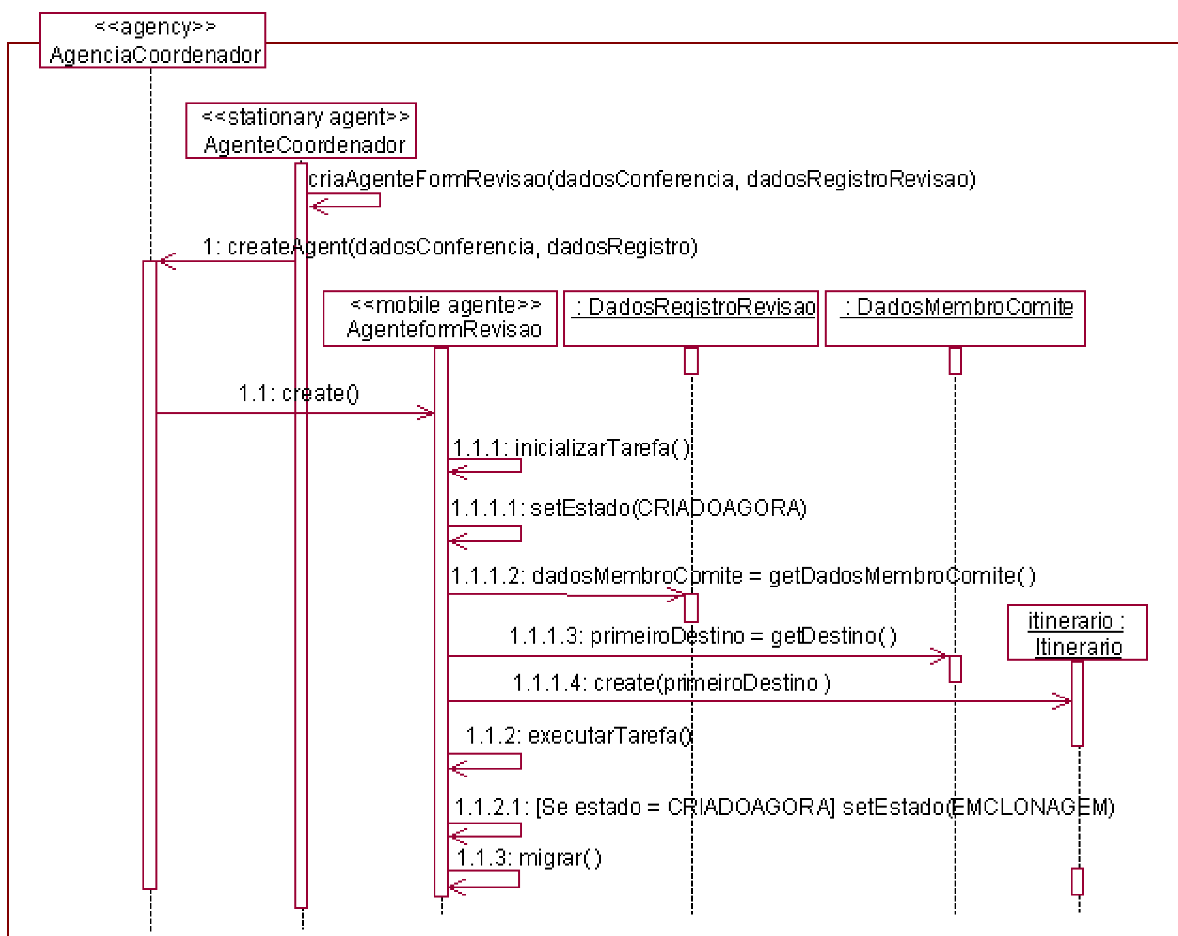

Figura 6.9: Diagrama de sequência da operação de sistema *gerarFormRevisao* – Parte 2

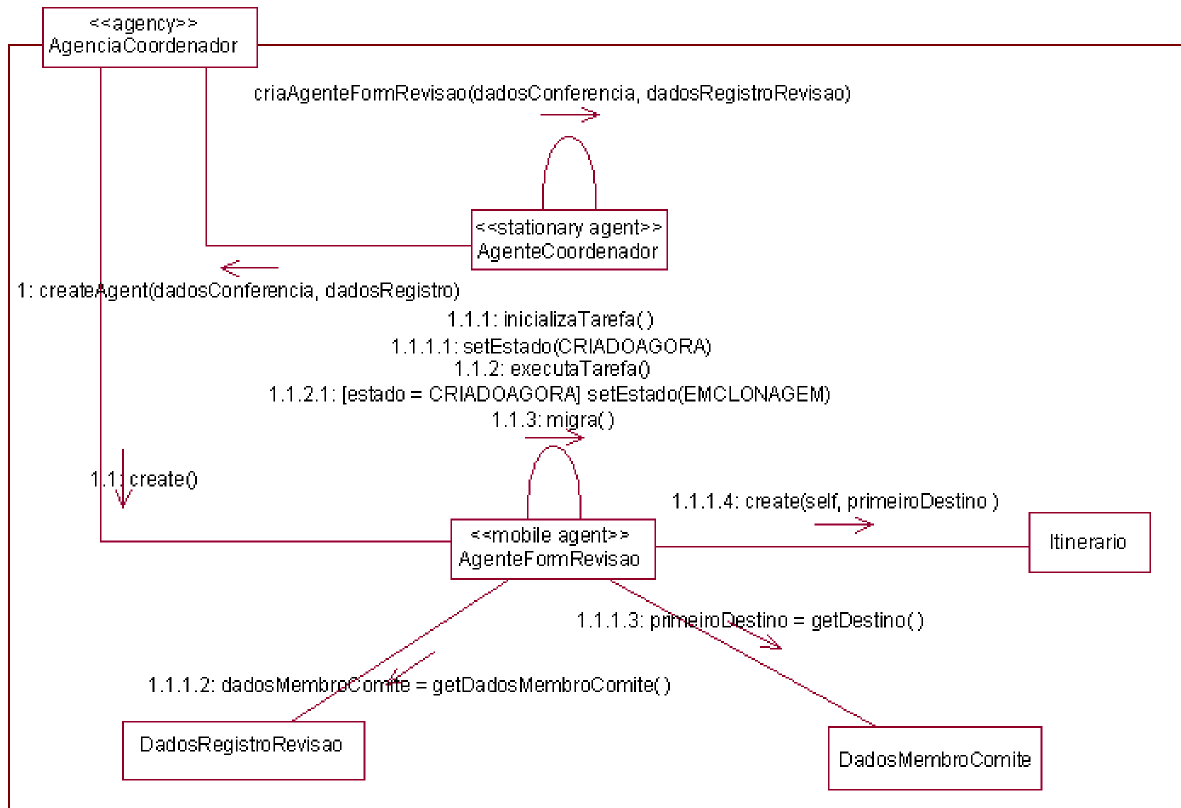

Figura 6.10: Diagrama de colaboração da operação de sistema *gerarFormRevisao* – Parte 2

Na Figura 6.11, terceira parte do diagrama de seqüência da operação de sistema *gerarFormRevisao,* o *AgenteFormRevisao* é transferido para a agência do membro de comitê (apesar de não estar ilustrado no diagrama de seqüência por razões de espaço, o objeto *Itinerario* também é transferido junto com o agente). Ao chegar em seu destino, o *AgenteFormRevisao* altera o valor do atributo estado para *EMDISTRIBUICAO* e gera os clones solicitados (*qntCopias* – 1). A Figura 6.12 ilustra o diagrama de colaboração correspondente.

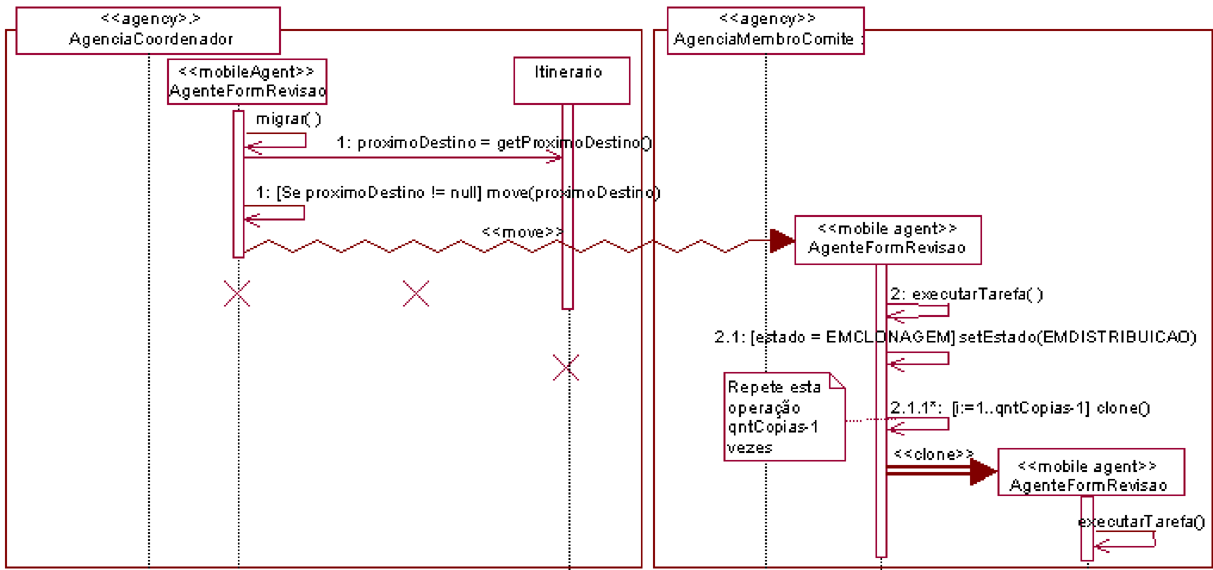

Figura 6.11: Diagrama de sequência da operação de sistema *gerarFormRevisao* – Parte 3

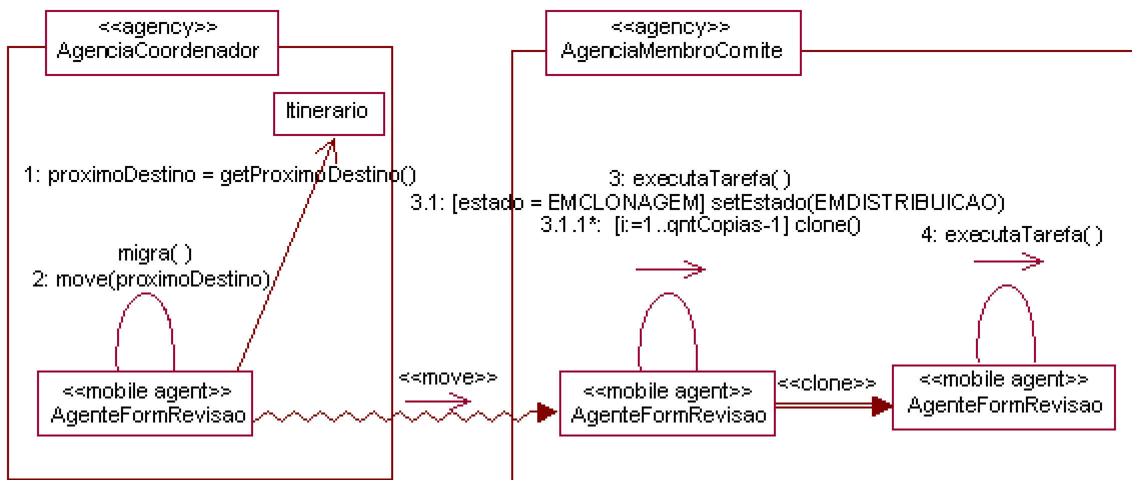

Figura 6.12: Diagrama de colaboração da operação de sistema *gerarFormRevisao* – Parte 3

6.4.1.2 Diagramas de interação da operação de sistema *redirecionarAgenteFormRevisao*

A Figura 6.13 ilustra o diagrama de seqüência da operação de sistema *redirecionarAgenteFormRevisao*. O membro de comitê (também pode ser um revisor) solicita que o *AgenteFormRevisao* se redirecione para a agência de um revisor e informa se ele deve retornar quando o processo de revisão tiver sido concluído. Caso o retorno do agente não tenha sido solicitado o endereço da agência do membro de comitê é removido do itinerário. O *AgenteFormRevisao* insere o próximo destino em seu itinerário, altera o valor do atributo

estado para *EMREVISAO* e usa o método *migrar* para se transferir para o próximo destino do seu itinerário.

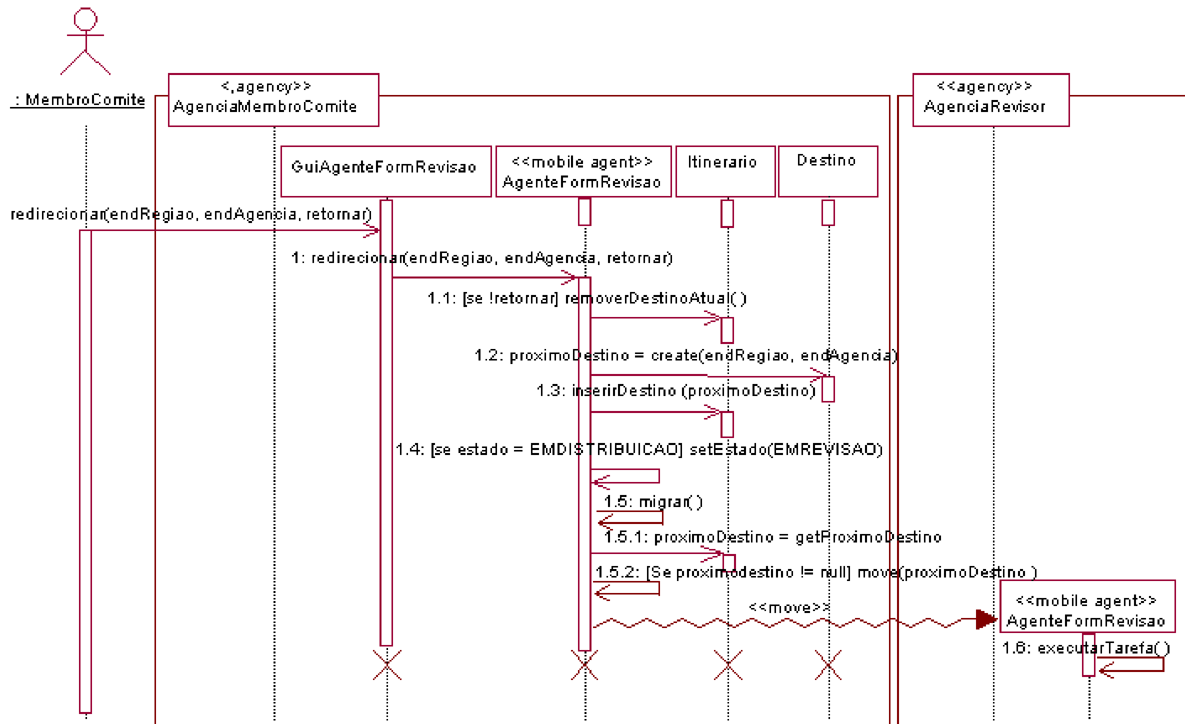

Figura 6.13: Diagrama de sequência da operação de sistema *redirecionarAgenteFormRevisao*

6.4.1.3 Diagramas de interação da operação de sistema *finalizarRevisao*

A Figura 6.14 ilustra o diagrama de seqüência da operação de sistema *fiinalizarRevisao.* Quando o ator revisor conclui a revisão do artigo, o *AgenteFormRevisao* altera o valor do atributo estado para *EMAPROVACAO* e chama o método *aprovarRevisao,* que é detalhado no diagrama de seqüência da Figura 6.15.

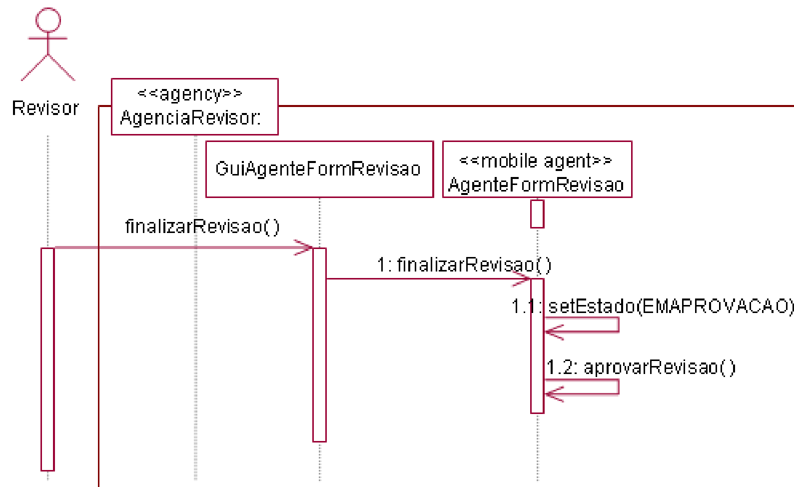

Figura 6.14: Diagrama de sequência da operação de sistema *finalizarRevisao*

6.4.1.4 Diagramas de interação da operação de sistema *aprovarRevisao*

A Figura 6.15 ilustra o diagrama de seqüência da operação de sistema *aprovarRevisao* para o caso específico em que ela é chamada pelo membro de comitê. Quando isso ocorre, o *AgenteFormRevisao* remove o endereço da localização atual do itinerário e usa o método migra para se transferir para o próximo destino, neste caso a agência do coordenador de programa. Ao chegar em seu destino, o *AgenteFormRevisao* entrega os dados coletados para o *AgenteCoordenador* (Figura 6.16).

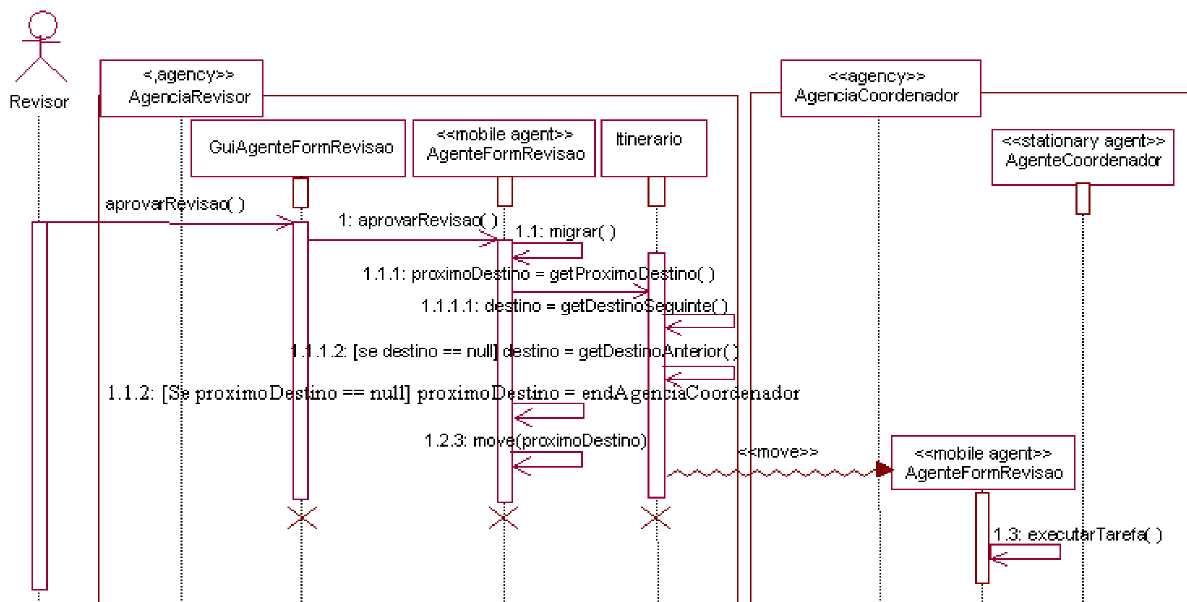

Figura 6.15: Diagrama de sequência da operação de sistema *aprovarRevisao* – Parte 1

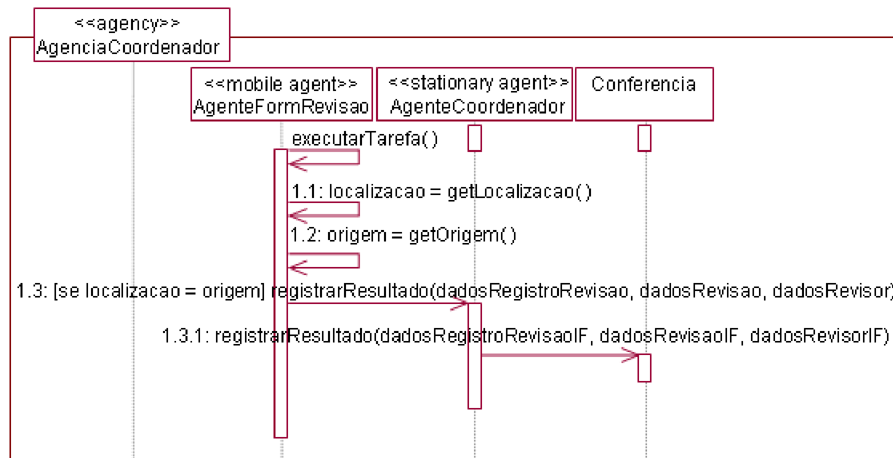

Figura 6.16: Diagrama de sequência da operação de sistema *aprovarRevisao* – Parte 2

6.4.2 Diagrama de Classes

Um diagrama de classes ilustra as especificações das classes e interfaces (classes que definem características de um componente disponíveis a clientes) em uma aplicação.

A elaboração de um diagrama de classes ocorre dentro da fase de projeto de um ciclo de desenvolvimento e é dependente dos seguintes artefatos:

- Diagramas de interação: é a partir deles que o projetista identifica as classes de software que participam da solução. Em diagramas de classe e diagramas de interação são criados em paralelo.
- Modelo conceitual: é a partir dele que o projetista adiciona detalhes às definições das classes.

O diagrama da Figura 6.17 ilustra as classes de objetos persistentes do nosso estudo de caso:

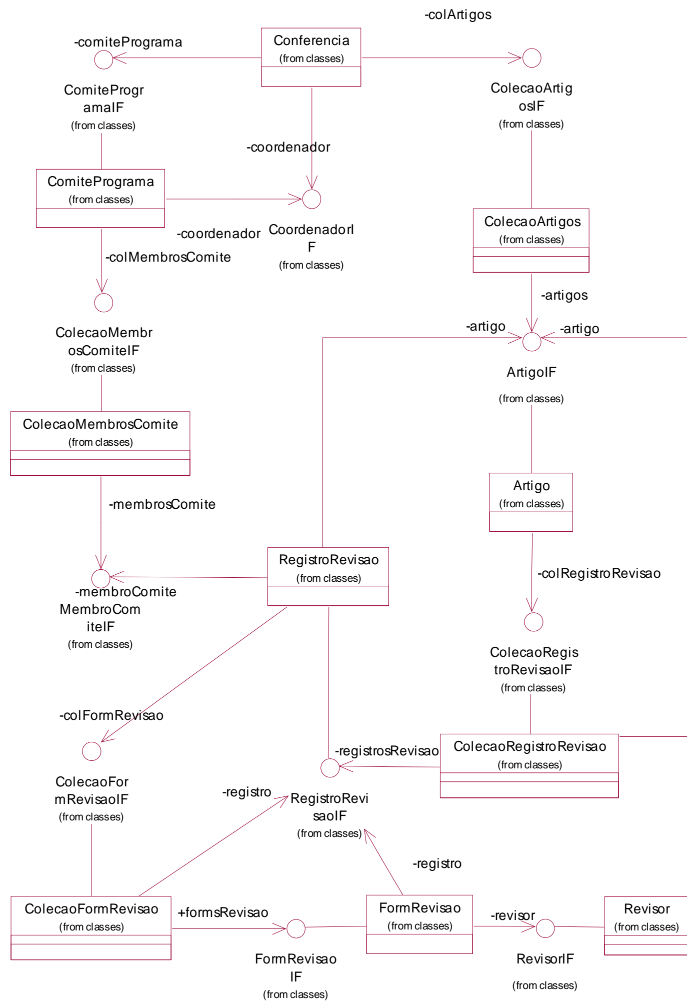

Figura 6.17: Diagrama de classes de objetos

A fim de representar agentes estacionários, agentes móveis e agências em um diagrama de classes, utilizaremos as seguintes extensões propostas por Klein et al [KRSW01]:

- O estereótipo <<*mobile agent*>>, derivado do metamodelo *class*, especifica a classe de objetos agentes móveis. Graficamente, em diagrama de classes, um agente móvel é representado da seguinte forma:

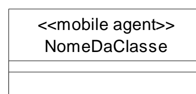

Figura 6.18: Representação Gráfica da Classe de um Agente Móvel em um Diagrama de Classes

- O estereótipo <<*agency*>>, também derivado do metamodelo *class*, especifica uma classe de objetos do tipo agência, ambiente de execução para os agentes móveis. Um objeto agência fornece aos agentes as funcionalidades e os serviços necessários para

serem executados. Graficamente, em um diagrama de classes, uma agência é representada da seguinte forma:

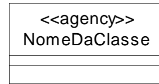

Figura 6.19: Representação Gráfica da Classe de uma Agência em um Diagrama de Classes

Já para representar a classe de um objeto estacionário, nós identificamos ainda a necessidade de mais uma extensão:

- O estereótipo <<*stationary agent*>>, derivado do metamodelo *class*. Graficamente, um agente estacionário é representado da seguinte forma:

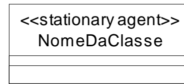

Figura 6.20: Representação Gráfica da Classe de um Agente Estacionário em um diagrama de Classes

A Figura 6.21 ilustra o uso destas extensões no diagrama de classes parcial para o nosso estudo de caso.

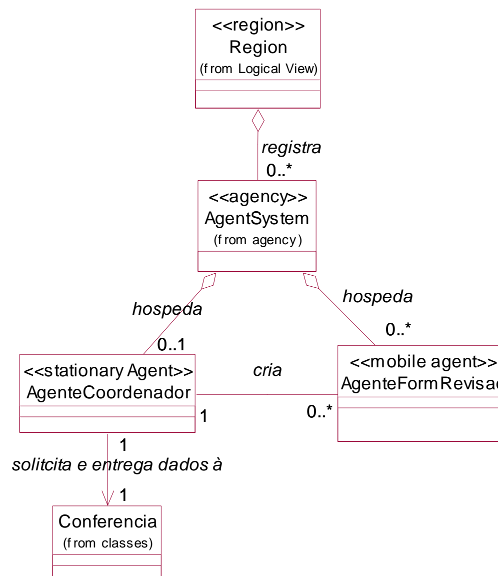

Figura 6.21: Diagrama de classes de agentes

Ao especificarmos uma classe de agente móvel precisamos avaliar que partes dos dados internos do agente devem ser mantidas quando ele migrar, ou seja, quais variáveis de instância farão parte do estado de dados do agente. Visto que o tamanho do estado de dados tem um alto impacto durante a migração, apenas um conjunto mínimo de variáveis

deve ser incluído. A seguir, dicas que ajudam o projetista a avaliar quais variáveis deverão estar incluídas no estado de dados dos agentes [IKV01]:

- Se, depois de cada migração, o valor de uma variável de instância é modificado antes que a variável seja lida pelo agente ou outros componentes, não é necessário transferir o valor antigo para a nova localização. Neste caso a variável deve ser transiente a fim de excluí-la do estado de dados. Especialmente se o agente cria a sua própria GUI, esta deve ser declarada transiente (ou definida dentro de um método). Além da redução do estado de dados, a principal razão para isto é que diversas classes GUI Java são dependentes do sistema operacional específico. Assim, se um agente migrar entre agências que rodam em diferentes sistemas operacionais e se este agente carregar um conjunto de classes GUI, falhas podem correr devido a problemas de incompatibilidade associados a estas classes.
- Se um agente instancia objetos que são (inteiramente ou parcialmente) não serializáveis, ocorrerá uma falha quando a agência tentar serializar o agente. Isto significa que, um agente que tenha alocado uma referência para um objeto não transiente, porém não serializável, não poderá migrar nem ser armazenado pelo serviço de persistência da agência. Para eliminar este problema, objetos não serializáveis devem ser transientes.

Lembre-se que todos os objetos referenciados por variáveis de instância, que fazem parte do estado de dados do agente, devem ser objetos inteiramente serializáveis.

O diagrama da Figura 6.22 ilustra o *AgenteFormRevisao* e os dados que ele transporta com ele durante a migração. As constantes *CRIADOAGORA*, *EMDISTRIBUICAO*, *EMREVISAO* e *EMAPROVACAO* são transientes, isto é, não fazem parte do estado de dados do agente. O atributo estado do *AgenteFormRevisao* é não transiente. O valor deste atributo é quem ajudará ao *AgenteFormRevisao* decidir que tarefa executar quando chegar ao seu destino. Observe, nos diagramas de seqüências apresentados na seção anterior, que sempre que o agente precisa realizar uma tarefa diferente da atual no próximo destino, ele altera o valor do atributo estado. Os outros atributos *dadosConferencia*, *dadosRegistroRevisao*, *dadosRevisao* e *itinerario*, que são referências para objetos serializáveis, são não transientes, ou seja, fazem parte do estado de dados do agente e são transportados com ele durante a sua migração.

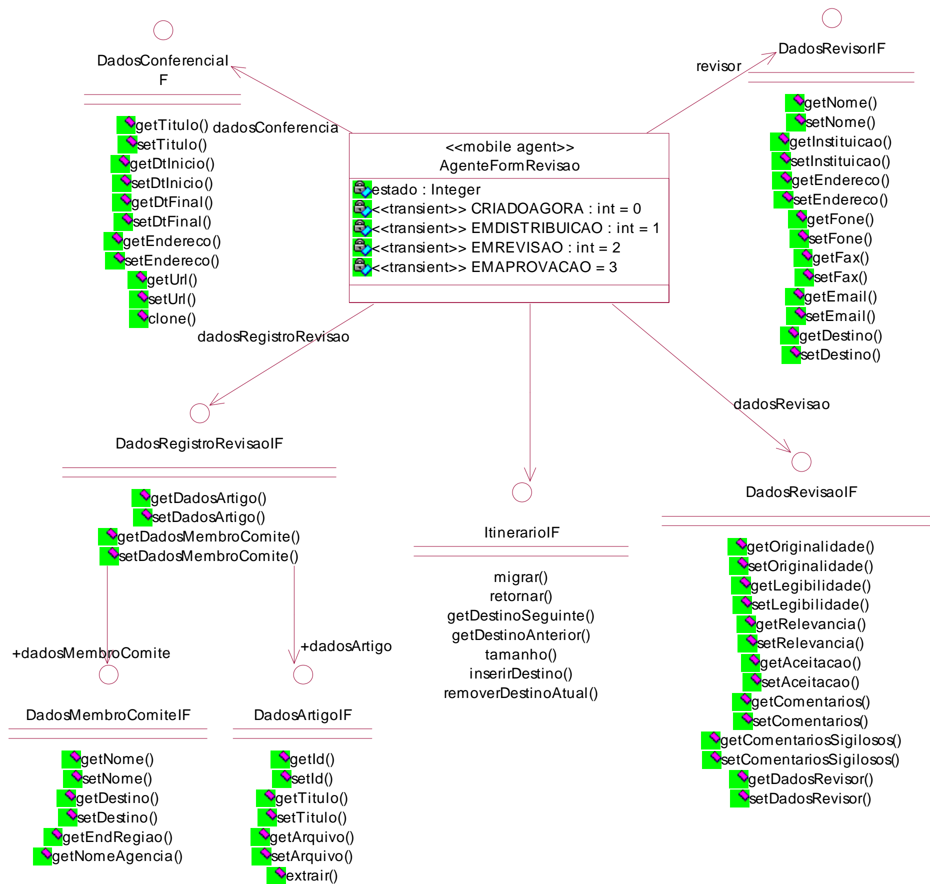

Figura 6.22: Diagrama de classes *AgenteFormRevisao*

Como já foi citado na Seção 6.5.1, uma adaptação do padrão de agentes móveis Mestre-Escravo Intinerante foi aplicado em nosso estudo de caso. Conforme apresentado na Figura 6.23, a classe *AgenteCoordenador* implementa a interface *AgenteMestreIF* e é responsável por criar o *AgenteFormRevisao* e registrar os dados da revisão entregues por ele. A classe *AgenteFormRevisao* representa o agente escravo itinerante e implementa, entre outros, os métodos *inicializarTarefa*( ), *executarTarefa*( ) e *migrar*( ). A classe *Itinerario* mantém a informação sobre o destino atual e implementa a interface *ItinerarioIF* que define os métodos: *getProximoDestino*( ), *getDestinoAtual*( ), *tamanho*( ), *inserirDestino*(*DestinoIF destino*) *e removerDestinoAtual*( ).

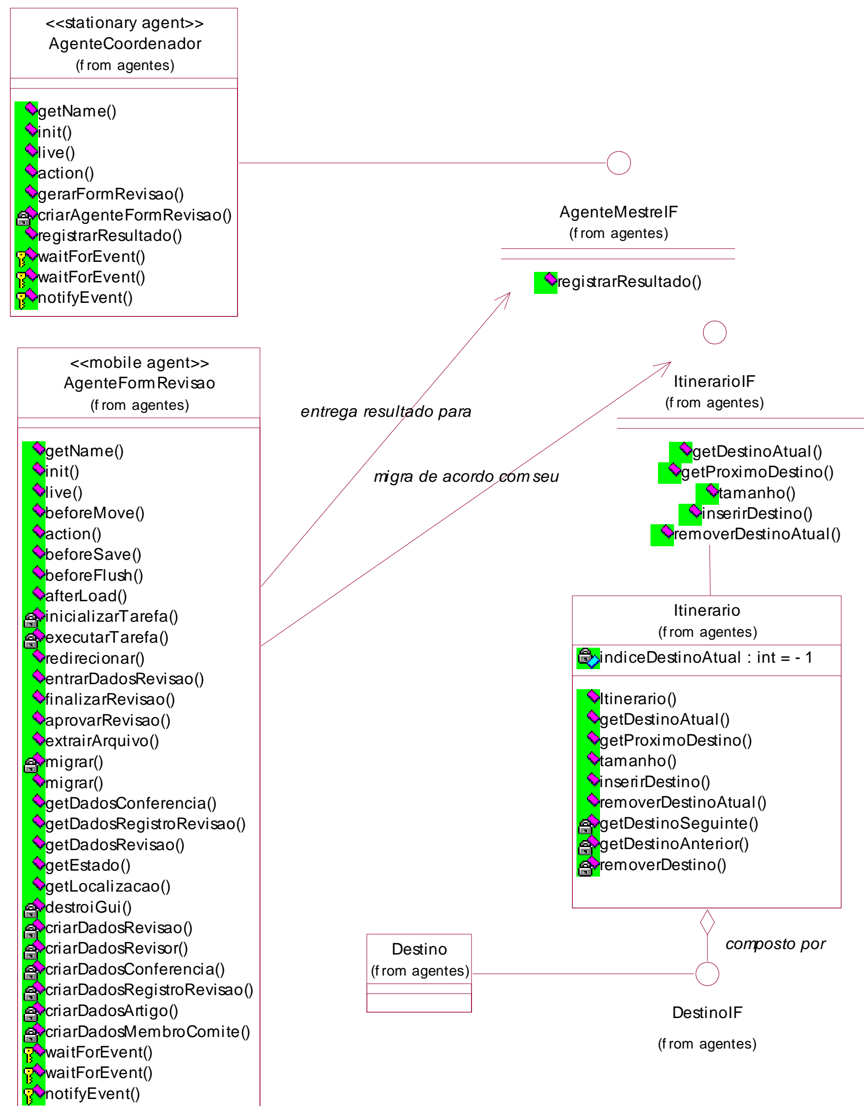

Figura 6.23: Diagrama de classes Mestre-Escravo Intinerante

6.4.3 Esquema de Banco de Dados

Este artefato deve ser gerado nos casos em que a aplicação requer armazenamento e recuperação de informações em um mecanismo de persistência.

As plataformas de agentes móveis apresentam mecanismos de persistência para os agentes que são transparentes para o desenvolvedor, mas que sugerimos que sejam utilizados apenas para os casos de ativação e desativação dos agentes. As informações carregadas pelos agentes, que precisam ser persistidas para uma futura recuperação devem ser especificadas como classes persistentes de objetos comuns.

Para que possamos persistir objetos em um banco de dados, seja ele orientado a objetos, objeto-relacional ou simplesmente relacional, será necessária a aplicação de técnicas de mapeamento a fim de gerar tabelas correspondentes as classes persistentes [BP98]. Todavia, já existem no mercado *frameworks* de persistência profissionais, como o *TopLink* (www.objectpeople.on.ca/toplink), que tornam isto transparente para o desenvolvedor.

6.5 Validação

Verificar a completude, a corretude e a consistência dos diagramas de interação com relação aos contratos de operações do sistema e ao modelo conceitual. O mesmo deve ser feito para os diagramas de classes com relação aos diagramas de interação e ao modelo conceitual, como também para o esquema de banco de dados com relação aos diagramas de classes.

6.6 Considerações finais

Este capítulo apresentou a fase de projeto detalhado independente de plataforma do modelo proposto para o desenvolvimento de aplicações baseadas em agentes móveis. O principal objetivo desta fase é produzir uma especificação lógica independente de uma plataforma de agentes móveis especifica e que reflita como o comportamento do agente é implementado para cumprir os requisitos funcionais e não funcionais do sistema.

Os principais artefatos a serem produzidos nesta fase consistem de diagramas de interação, diagramas de classes e esquema de banco de dados (caso haja necessidade de persistência de informação).

O modelo proposto sugere a utilização da extensão de UML proposta por Klein et al [KRSW01] a fim de modelar regiões, agências, agentes estáticos e agentes móveis nos diagramas de classes e também mobilidade e clonagem de agentes nos diagramas de interação.

Os diagramas de interação definem a forma como os agentes e objetos interagem e devem ser elaborados através da aplicação de padrões GRASP (*General Responsability Assignment Software Patterns*) [Lar02], padrões de projeto O.O. [GHJV95] e padrões de agentes móveis [LO98]. Na Seção 6.5.1, descrevemos como o padrão de agentes móveis Mestre-Escravo Itinerante (*Master-Slave Revisited)* proposto por Lange et al [LO98] foi adaptado e aplicado em nosso estudo de caso.

Os diagramas de classes definem as especificações de classes e interfaces em uma aplicação. Na Seção 6.5.2 dicas para auxiliar o projetista a avaliar quais variáveis deverão estar incluídas no estado de dados dos agentes foram apresentadas, visto que o tamanho deste tem um alto impacto durante a migração do agente.

Nos casos em que a aplicação requer armazenamento e recuperação de informações em um mecanismo de persistência, um esquema de banco de dados deve ser gerado e isto pode ser feito da forma convencional. Apesar das plataformas de agentes móveis apresentarem mecanismos de persistência para os agentes que são transparentes ao desenvolvedor, sugerimos que estes sejam utilizados apenas para ativação e desativação dos agentes. As informações carregadas pelos agentes, que precisam ser persistidas para uma futura recuperação devem ser especificadas como classes persistentes de objetos comuns.

Como já foi dito anteriormente, por questões de reuso do projeto no caso de haver uma mudança na escolha da plataforma de agentes móveis ou no caso da aplicação requisitar implementações em plataformas diferentes, resolvemos dividir a fase de projeto detalhado em duas, sendo uma independente e a outra dependente de plataforma. A primeira fase foi apresentada no presente capítulo e a segunda, fase de projeto detalhado dependente de plataforma, será apresenta no Capítulo 7.

7 PROJETO DETALHADO DEPENDENTE DE PLATAFORMA

Este capítulo apresenta a fase de projeto detalhado dependente de plataforma que corresponde ao nível de microarquitetura apresentado em [TOH99]. Nesta fase, padrões de projeto de agentes móveis determinados pela plataforma na qual a aplicação será desenvolvida serão aplicados e novas classes, interfaces e relacionamentos deverão ser adicionados as especificações lógicas do sistema.

A seguir, o objetivo da fase de projeto detalhado dependente de plataforma, bem como seus artefatos de entrada, as tarefas a serem realizadas, os artefatos a serem produzidos e os critérios de validação serão descritos.

7.1 Objetivo

Estender à especificação lógica produzida na fase de projeto detalhado independente de plataforma a fim de endereçar as características de projeto e implementação relacionadas a uma plataforma de agentes móveis específica.

7.2 Entradas

Os artefatos gerados na fase de projeto dependente de plataforma dependem dos artefatos produzidos nas fases de análise, de projeto arquitetural e de projeto detalhado independente de plataforma.

7.3 Tarefas

Nesta fase devemos incluir as novas classes e relacionamentos que surgiram devido à adoção dos padrões de projeto determinados pela plataforma escolhida, na especificação lógica do sistema. Desta forma, as atividades desta fase consistem em:

- Adotar os padrões de projeto relativos à implementação de características de comunicação, segurança e confiabilidade específicos da plataforma. Por exemplo, a plataforma de agentes móveis Grasshopper permite a comunicação entre agentes de forma transparente de localização, todavia obriga ao programador a adotar o padrão de comunicação Proxy para que isso seja possível (ver detalhes na Seção 7.4.1);
- No caso da plataforma escolhida suportar apenas migração fraca, adotar padrões de projeto que a simulem migração forte;

- Refinar os diagramas de interação e de classes, adicionando os detalhes que surgiram devido à adoção dos padrões;
- Nos casos em que as plataformas são baseadas em Java, recomendamos fortemente que uma documentação Javadoc seja gerada.

7.4 Artefatos

7.4.1 Diagramas de Interação Dependentes de Plataforma

Os diagramas de interação dependentes de plataforma refinam os diagramas de interação elaborados na fase de projeto detalhado independente de plataforma, adicionando os novos objetos e comportamentos que surgem após a aplicação de padrões de agentes móveis específicos da plataforma de desenvolvimento escolhida.

Como já foi dito anteriormente, a plataforma de agentes móveis escolhida para o desenvolvimento do nosso estudo de caso foi a Grasshopper. Esta plataforma oferece serviços de comunicação que podem ser usados de acordo com padrões de comunicação específicos.

Grasshopper permite tanto a comunicação local quanto a remota entre seus componentes (agentes, agências e regiões) e aplicações ou objetos externos. Uma propriedade desejável a sistemas de agentes móveis é a transparência de localização. Em outras palavras, a entidade que deseja se comunicar com um agente móvel não precisa se preocupar com a sua localização. A fim de prover esta característica a plataforma Grasshopper utiliza-se do padrão de comunicação Proxy. O Proxy consiste em uma entidade intermediária na comunicação entre cliente e servidor e é responsável pelo estabelecimento da conexão, sendo da sua responsabilidade identificar a localização do servidor que pode ser um agente móvel. Em uma comunicação cliente-servidor na plataforma Grasshopper o cliente sempre acessa o proxy do servidor através de chamadas locais. Então, o proxy encaminha as chamadas por meio do canal de comunicação para o servidor.

A fim de usar o serviço de comunicação Grasshopper faz-se necessário seguir os seguintes passos:

- Criar a interface do componente servidor (interface servidor), que é quem define quais métodos poderão ser acessados pelo cliente. As seguintes entidades podem ser servidores: agentes, agências, regiões e aplicações ou objetos externos. Todavia, a interface do servidor só precisa ser criada nos casos em que o servidor for um agente

ou uma entidade externa, pois a plataforma Grasshopper já fornece as interfaces para a agência e para o serviço de domínio de agências (registro de região).

- Implementar a entidade servidora que deve oferecer ao menos um método público para o serviço de comunicação. Todos os métodos que serão acessados via o serviço de comunicação terão que estar incluídos na interface Java implementada pelo objeto servidor.
- Implementar a entidade cliente (agente ou aplicação/objeto externo) que deve criar um proxy da entidade servidora. Este proxy fornece todos os métodos públicos que foram definidos na interface do servidor. Devido às capacidades de reflexão do JDK 1.3 ou superior é possível para entidade cliente criar o proxy do servidor dinamicamente em tempo de execução.

Em nosso estudo de caso, a *GuiAgenteCoordenador* e a *GuiFormRevisao* se comunicam respectivamente com o *AgenteCoordenador* e com o *AgenteFormRevisao* para repassar as solicitações do usuário. Por sua vez, o *AgenteCoordenador,* se comunica com o objeto externo *Conferencia* em duas situações: a primeira para obter os dados sobre a conferência que o *AgenteFormRevisao* precisará levar e a segunda para registrar os dados da revisão entregues a ele pelo *AgenteFormRevisao*. O *AgenteCoordenador* também interage com a sua agência para solicitar a criação do *AgenteFormRevisao*. Ao retornar para a máquina do coordenador de programa, o *AgenteFormRevisao* irá se comunicar com o *AgenteCoordenador* para entregar os dados da revisão. Devido ao uso do padrão Proxy, objetos do tipo *proxys* foram adicionados ao diagramas independentes de plataforma apresentados na Seção 6.5.1 gerando os diagramas dependentes de plataforma das Figuras 7.1, 7.2, 7.3, 7.4, 7.5, 7.6 e 7.7.

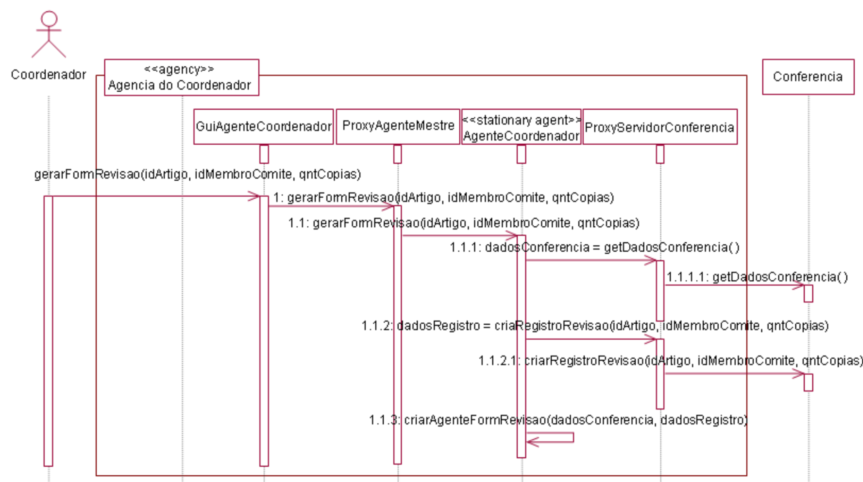

Figura 7.1: Diagrama de sequência dependente de plataforma da operação *gerarFormRevisao* – Parte 1

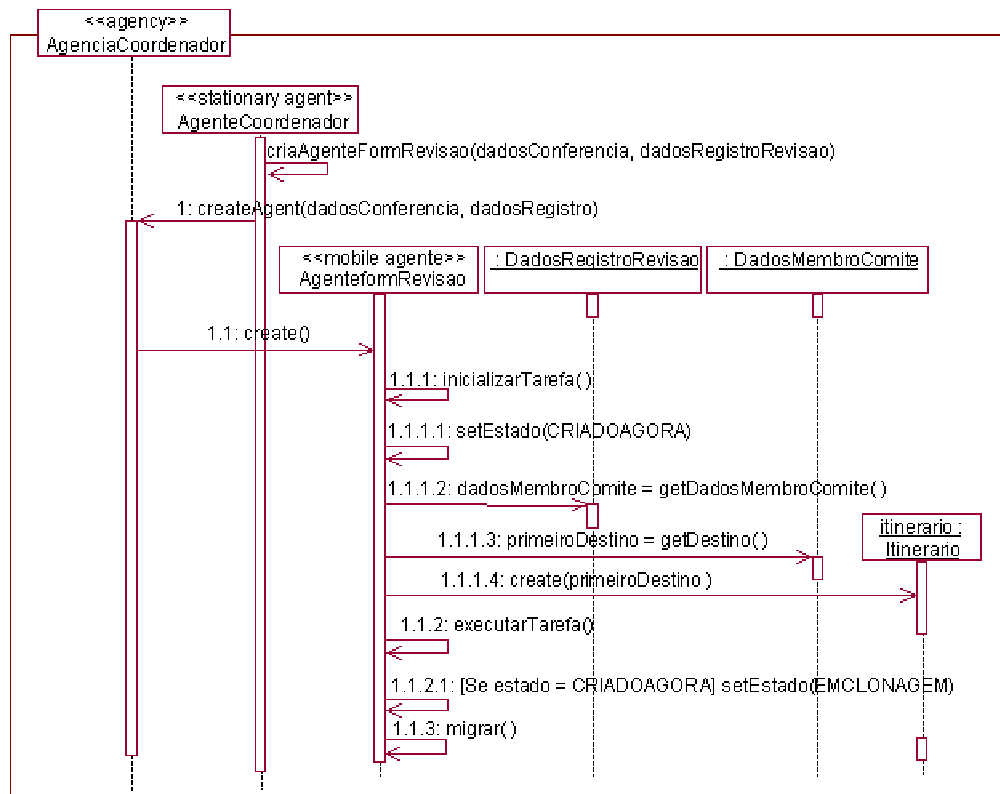

Figura 7.2: Diagrama de sequência dependente de plataforma da operação *gerarFormRevisao* – Parte 2

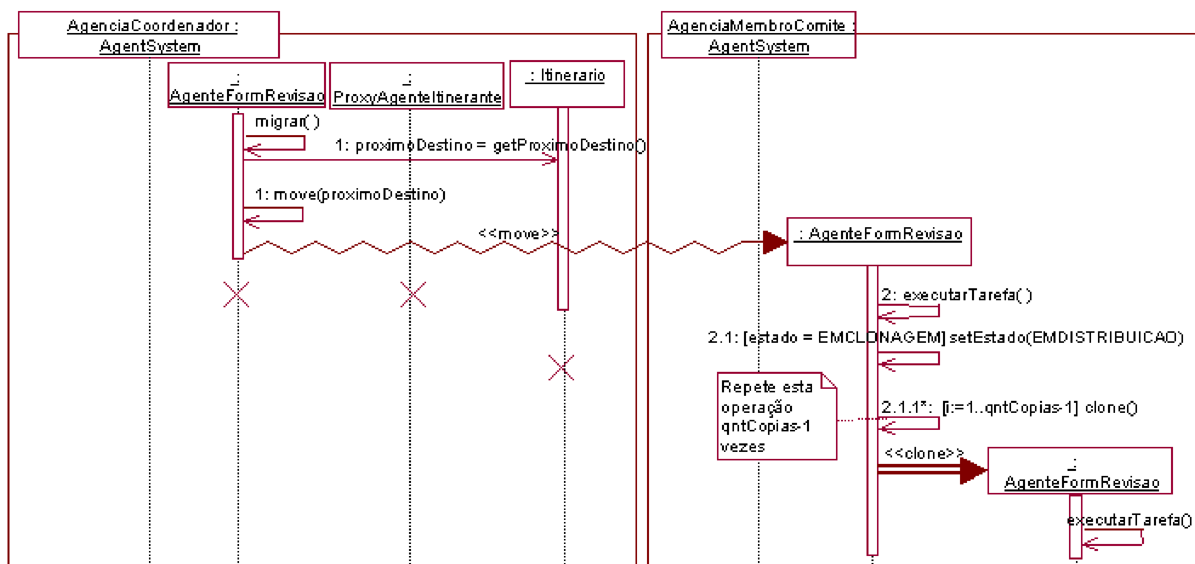

Figura 7.3: Diagrama de sequência dependente de plataforma da operação *gerarFormRevisao* – Parte 3

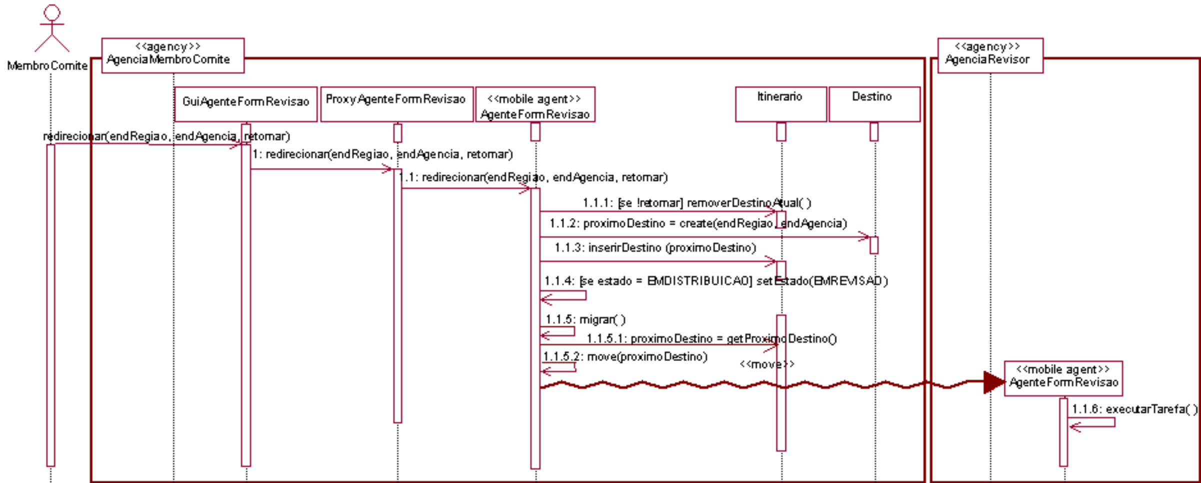

Figura 7.4: Diagrama de sequência dependente de plataforma da operação *redirecionarAgenteFormRevisao*

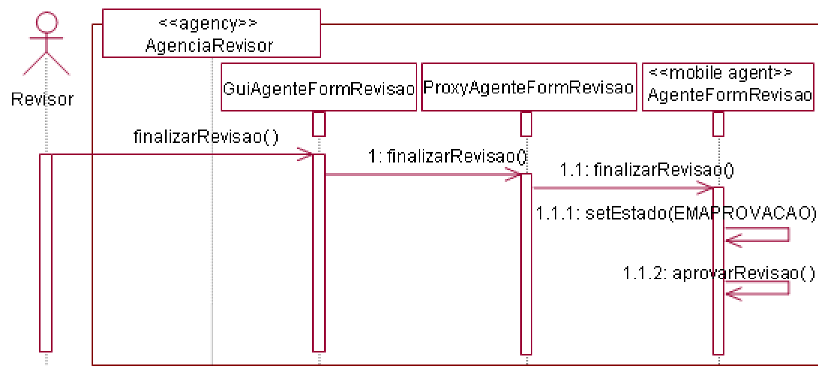

Figura 7.5: Diagrama de sequência dependente de plataforma da operação *finalizarRevisao*

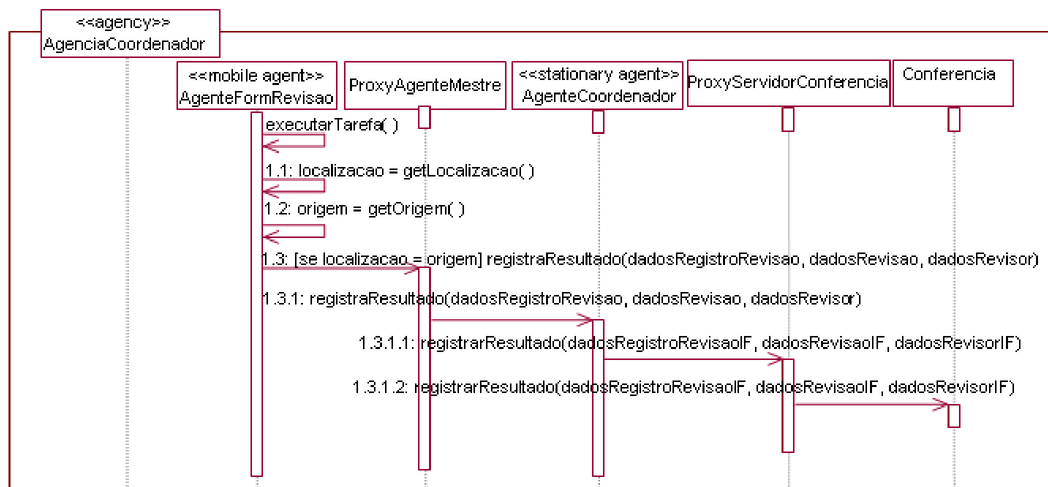

Figura 7.6: Diagrama de sequência dependente de plataforma da operação *aprovarRevisao* – Parte 2

7.4.2 Simulação da Migração Forte

A fim de simularmos a migração forte na plataforma de agentes móveis Grasshopper, que suporta apenas migração fraca, declaramos uma variável de instância não transiente `estado` e dividimos o método `executarTarefa()` do *AgenteFormRevisao*, que é chamado dentro do seu método `live()`, em blocos de execução diferentes através de comandos condicionais. Para determinar que bloco deve executar, o agente analisa o valor da variável `estado`.

Após executar o método `executarTarefa()`, o *AgenteFormRevisao* migra para o próximo destino do seu itinerário. Observe o diagrama de atividades do método `executarTarefa()` na figura 7.7.

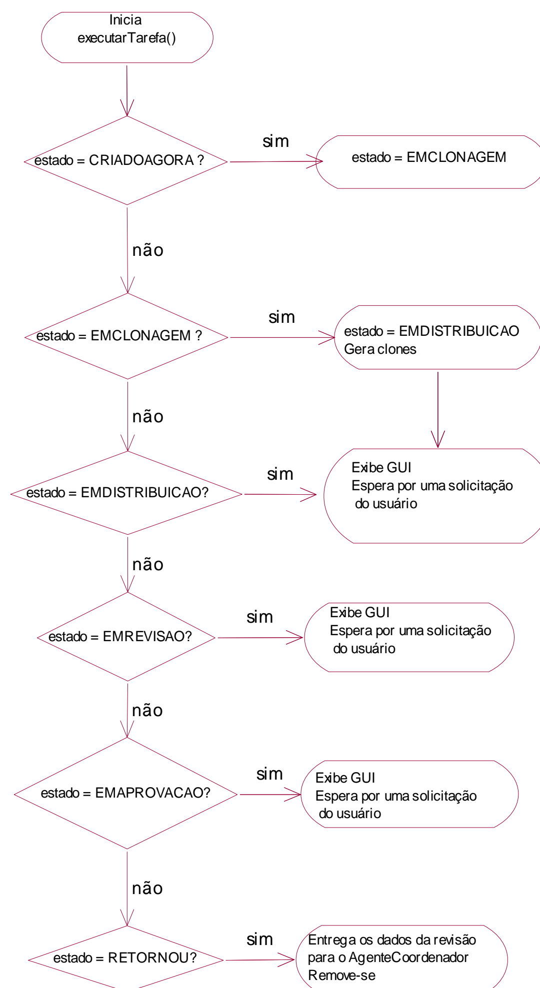

Figura 7.7: Diagrama de atividades do método `executarTarefa()` da classe *AgenteFormRevisao*

7.4.3 Diagrama de Classes

Os diagramas de classe dependentes de plataforma refinam os diagramas de classe elaborados na fase de projeto detalhado independente de plataforma adicionando as novas classes, interfaces e associações que surgem após a aplicação de padrões de agentes móveis específicos da plataforma de desenvolvimento adotada.

As Figuras 7.8, 7.9 e 7.10 apresentam os diagramas de classes com as novas interfaces e classes geradas devido à aplicação do padrão Proxy.

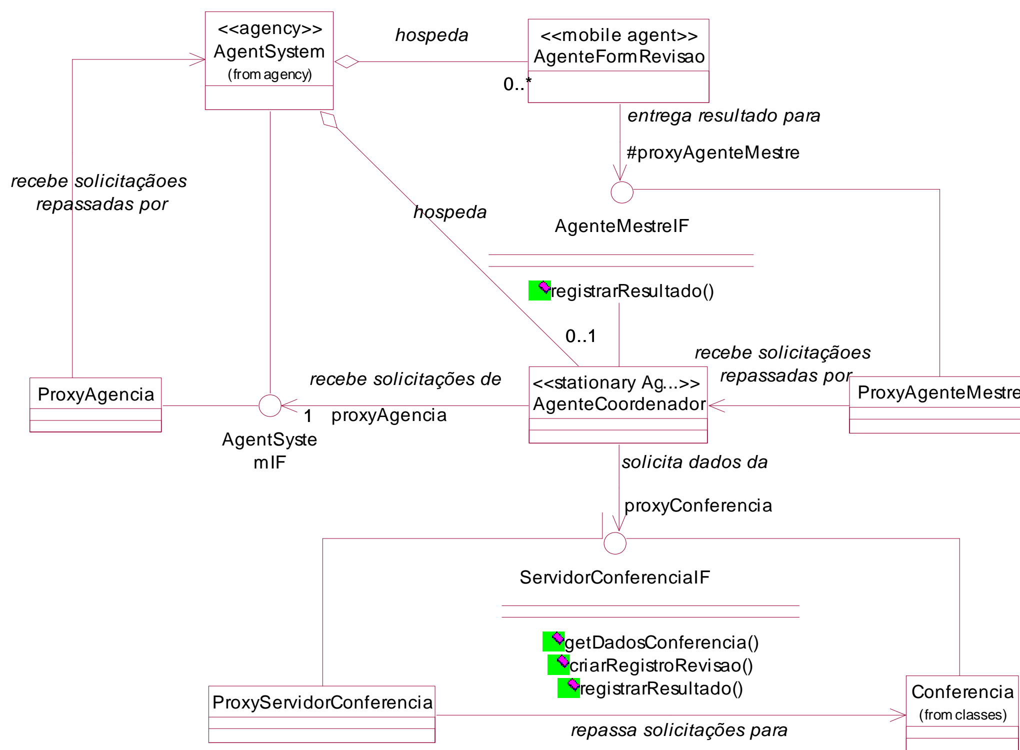

Figura 7.8 : Diagrama de classes dependente de plataforma

Na Figura 7.8 temos as seguintes classes adicionais:

- ***ProxyAgencia.*** Descreve o proxy da agência a ser utilizado pelo *AgenteCoordenador* ao solicitar a criação do *AgenteFormRevisao*. Esta classe implementa a interface ***AgentSystemIF*** fornecida pela plataforma Grasshopper, disponibilizando todas as funcionalidades da agência.
- ***ProxyConferencia***. Descreve o proxy do objeto externo Conferencia a ser utilizado pelo *AgenteCoordenador* para solicitar dados da conferência e para registrar os dados da revisão entregues a ele pelo *AgenteFormRevisao*. Esta classe implementa a interface *ServidorConferenciaIF* que disponibiliza os métodos

getDadosConferencia( ), *criarRegistroRevisao(int idArtigo, idMembroComite, int qntCopias)* e *registrarResultado(DadosRegistroRevisao dadosRegistroRevisao, dados)*.

- ***ProxyAgenteMestre***. Descreve o proxy do *AgenteCoordenador* a ser utilizado pelo *AgenteFormRevisao* para entregar os dados da revisão. Esta classe implementa a interface ***AgenteMestreIF*** do padrão Mestre Escravo Itinerante disponibilizando o método *registrarResultado*(*DadosRegistroRevisao dadosRegistroRevisao, dados*).

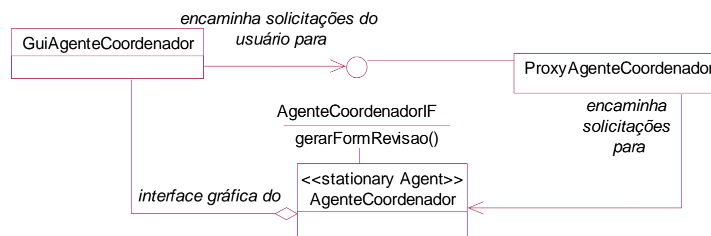

Figura 7.9 : Diagrama de classes dependente de plataforma do AgenteCoordenador

A Figura 7.9 apresenta a classe adicional ***ProxyAgenteCoordenador*** que descreve o proxy do *AgenteCoordenador* utilizado pela *GuiAgenteCoordenador* para repassar as solicitações do usuário. Esta classe implementa a interface ***AgenteCoordenadorIF*** que disponibiliza o método *gerarFormRevisao(int idArtigo, int idMembroComite, int qntCopias)*.

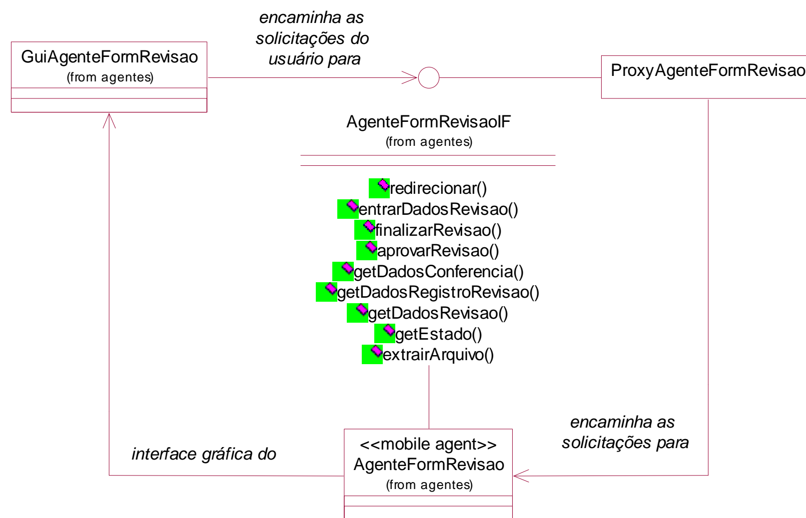

Figura 7. 10: Diagrama de classes dependente de plataforma do *AgenteFormRevisao*

As seguintes classes adicionais são apresentadas na Figura 7.10:

- ***ProxyAgenteFormRevisao.*** Descreve o proxy do *AgenteFormRevisao* utilizado pela *GuiAgenteFormRevisao* para repassar as solicitações do usuário. Esta classe implementa a interface ***AgenteFormRevisaoIF*** que disponibiliza os métodos *redirecionar(*String endRegiao, String nomeAgencia, boolean retornar*), extrairArtigo(String diretorio), entrarDados(String nomeRevisor, String instituicaoRevisor, String endRevisor, String foneRevisor, String faxRevisor, String emailRevisor, int originalidade, int legibilidade, int relevancia, int aceitação, int comentarios, int comentariosSigilosos), fimRevisao( )* e *aprovarRevisao( )*.

7.5 Validação

Verificar a completude, a corretude e a consistência dos diagramas de interação e de classes dependentes de plataforma com os diagramas de interação e de classes independentes da plataforma. A mesma verificação deve ser feita para a documentação Javadoc com relação aos diagramas de classe dependentes de plataforma.

7.6 Considerações Finais

Este capítulo apresentou a fase de projeto detalhado dependente de plataforma do modelo proposto para o desenvolvimento de aplicações baseadas em agentes móveis. Esta fase corresponde ao nível de microarquitetura apresentado em [TOH99] e o seu principal objetivo é estender a especificação lógica produzida na fase de projeto independente de plataforma a fim de endereçar as características de projeto e implementação relacionadas a uma plataforma de agentes móveis específica.

Nesta fase, novas classes e relacionamentos surgem devido à adoção de padrões de projeto determinados pela plataforma de agentes móveis escolhida e conseqüentemente os diagramas de classe e de interação precisam ser refinados. Em nosso estudo de caso, a plataforma de agentes móveis Grasshopper foi utilizada e o padrão de comunicação Proxy teve que ser adotado para permitir a comunicação entre agentes, agências, regiões e entidades externas à plataforma. Os diagramas de seqüência e de classes gerados devido a adoção deste padrão foram apresentados nas Seções 7.5.1 e 7.5.2.

Visto que esta fase isola todas as decisões de baixo nível dependentes de plataforma, apenas ela precisará ser refeita no caso de ocorrer uma mudança na escolha da plataforma de agentes móveis durante ou após execução do projeto.

No Capítulo 8, as conclusões e sugestões para trabalhos futuros serão apresentadas.

8 CONCLUSÕES

Métodos e técnicas O.O. (Orientadas a Objetos) existentes para o desenvolvimento de aplicações baseadas em componentes distribuídos não são suficientes para cobrir todos os aspectos de mobilidade na modelagem, projeto e verificação de sistemas baseados em agentes móveis [WJK99, OPB00, MKC01]. Pudemos verificar a ineficiência destes métodos e técnicas ao tentar aplicá-los no desenvolvimento do nosso estudo de caso. Durante a fase de análise, não tínhamos idéia de como deveríamos abordar o conceito de agentes. Ao elaborarmos o projeto arquitetural não sabíamos que decisões específicas para sistemas baseados em agentes móveis deveriam ser consideradas e durante a fase de projeto detalhado, sentimos muita dificuldade para modelar os conceitos relacionados a tecnologia de agentes móveis utilizando a UML padrão. Os diagramas de interação ficavam muito poluídos e difíceis de entender. Resolvemos então adotar uma metodologia para o desenvolvimento de aplicações baseadas em agentes móveis, todavia encontramos apenas algumas abordagens para atividades específicas do desenvolvimento e foram estas as razões que nos motivaram a propor um modelo que auxiliasse no desenvolvimento de aplicações baseadas em agentes móveis.

O modelo proposto neste trabalho não sugere um processo totalmente inovador, tendo sido construído a partir de técnicas e métodos amplamente utilizados por projetistas de software, tais como o processo unificado, UML e padrões de projeto, o que facilita o seu aprendizado e aplicação. Este modelo adapta o processo de desenvolvimento de software proposto por Larman [Lar02] adicionando considerações sobre aspectos relevantes à construção de aplicações baseadas em agentes móveis especificamente nas fases de análise e projeto. Fases como a de testes, manutenção e gerência de processo precisam ser devidamente endereçadas e adicionadas ao modelo para que este possa se tornar uma metodologia completa.

Na fase de análise definida no modelo proposto, conceitos de agentes podem ser introduzidos gradualmente a partir das especificações dos casos de usos, desde que eles sejam intrínsecos ao domínio do problema. Desta forma, conceitos de agentes e objetos passam a figurar lado a lado no modelo conceitual. Para investigar o comportamento do sistema, focando-se apenas no que ele deve fazer e não no como, diagramas de seqüência, diagramas de atividades e contratos de operações do sistema devem ser produzidos. Nos diagramas de seqüência e nos diagramas de atividades, que devem ser gerados a partir das especificações

dos casos de usos, agentes que interagem diretamente com os atores são ilustrados, representado uma parte do sistema. Quanto aos contratos, estes descrevem o que cada operação do sistema promete cumprir através das pós-condições, ou seja, o que muda no estado do sistema após a execução da operação. Em um sistema baseado em agentes móveis, além das categorias convencionais de pós-condições (criação e destruição de instâncias, modificação de atributos, associações formadas e quebradas) lembramos que também poderão ser identificadas as seguintes outras: migração, quando o agente migra para outra localização; desativação, quando o agente é desativado; e ativação, quando o agente é ativado. Um ponto negativo nos contratos do modelo proposto é a sua sintaxe livre que dificulta o entendimento e a manutenção dos mesmos, este problema poderia ser solucionado através da adoção de notações formais que também daria suporte a geração automática de testes.

A fase de projeto consiste em uma extensão do modelo de análise visando especificar em detalhes como a solução será implementada. A fim de permitir o uso efetivo de padrões de projeto de agentes móveis e promover o reuso, a fase de projeto foi dividida em três subfases: projeto arquitetural, projeto independente de plataforma e projeto dependente de plataforma. Esta divisão é baseada nas idéias apresentadas por Tahara et al [TOH99], onde os sistemas baseados em agentes são projetados de acordo com os seguintes níveis: macroarquitetura – projeto de alto nível independente de plataforma – e microarquitetura – projeto detalhado e comportamento de agentes especializado em uma plataforma específica. Em nosso modelo, as fases de projeto arquitetural e de projeto dependente de plataforma correspondem aos níveis de projeto de macroarquitetura e de microarquitetura propostos por Tahara et al. A principal diferença entre os dois modelos é que nós incluímos a fase de projeto independente de plataforma que trata de todo o projeto lógico da aplicação sem entrar em detalhes relacionados a uma plataforma de agentes móveis específica. Esta divisão foi motivada pela possibilidade de reuso do projeto, pois caso haja alteração na escolha da plataforma de agentes móveis apenas a segunda fase do projeto detalhado precisará ser refeita. Além disso, um outro diferencial da fase de projeto detalhado é que os diagramas de classe e os diagramas de interação, através da utilização da extensão de UML proposta por Klein et al [KRSW01], são capazes de representar regiões, agências, agentes estáticos, agentes móveis, como também mobilidade e clonagem de agentes.

Apesar das vantagens listadas, a divisão da fase de projeto em duas apresenta a desvantagem de aumentar a quantidade de artefatos a serem gerados tornando inviável dar manutenção e garantir a consistência dos mesmos sem uma ferramenta de suporte adequado. Para produzir os diagramas de classe e de interação do nosso estudo de caso utilizamos a

ferramenta Rational Rose Enterprise Edition (http://www.rational.com), todavia não conseguimos adicionar a ela todas as extensões de UML proposta por Klein, assim fomos obrigados a adicioná-las aos diagramas de forma manual através da sua edição no Word.

Com o propósito de refinar e analisar a viabilidade do modelo proposto realizamos um estudo de caso através do desenvolvimento de uma aplicação para a Internet sugerida por Cardelli em [Car99]: Sistema de Apoio às Atividades de Comitês de Programa em Conferências. Devido a restrições de tempo apenas uma iteração do modelo foi executada abordando os seguintes casos de uso: Gerar Formulário de Revisão; Redirecionar Formulário de Revisão e Revisar Formulário de Revisão. A análise e projeto da aplicação foi realizada conforme o modelo proposto e os principais artefatos produzidos foram apresentados nesta dissertação. A aplicação foi implementada na plataforma de agentes móveis Grasshopper a partir das especificações lógicas do projeto dependente de plataforma. Técnicas de testes para aplicações baseadas em agentes móveis não foram encontradas, mas sugerimos que ao menos testes de unidade, testes funcionais e testes de sistema sejam realizados.

Outros estudos de casos também foram produzidos. Uma versão preliminar do modelo proposto foi utilizada pelos alunos do curso de desenvolvimento de sistemas de telecomunicações oferecido pelo Programa de Capacitação Tecnológica - PCT/DSC/UFCG e pela aluna de iniciação científica, Vivianne Medeiros, no desenvolvimento de estudos de caso para o domínio de telecomunicações.

Além do mais, o modelo proposto será utilizado por um aluno de mestrado do grupo de pesquisa de Sistemas de Informações e Banco de Dados, Philip Stephen Medcraft, orientado pelo doutor Ulrich Schiel, no trabalho de dissertação intitulado "Aplicando Web Semântica para a promoção de interoperabilidade entre centrais de corporações comerciais distribuídas".

8.1 Contribuições

A principal contribuição deste trabalho é a proposta de um modelo para o desenvolvimento de aplicações baseadas em agentes móveis construído a partir de técnicas e métodos amplamente utilizados por projetistas de software, tais como o processo unificado, UML e padrões de projeto, bem como a experiência do desenvolvimento do estudo de caso na plataforma Grasshopper, os quais servirão de referência para desenvolvimentos futuros de aplicações deste tipo.

Além da contribuição principal, temos ainda: a validação e extensão das idéias apresentadas por Tahara et al em [TOH99] que influenciaram a fase de projeto do nosso modelo.

Uma outra contribuição deste trabalho é que as notações de UML propostas por Klein et al em [KRSW01] também foram validadas através da sua aplicação em nosso estudo de caso, o que nos permitiu identificar a necessidade de mais uma extensão para representar agentes estacionários nos diagramas de classes e nos diagramas de interação.

Além do mais, esta dissertação é um trabalho precursor na área de metodologias para o desenvolvimento de aplicações baseadas em agentes móveis dentro do grupo de Engenharia de Software do Departamento de Sistemas e Computação da UFCG. Assim, motivados pela investigação no tema e pelos inúmeros problemas em aberto e potencialidades, projetos de pesquisa em desenvolvimento de software baseados em agentes móveis encontram-se em elaboração e passam a fazer parte dos pontos centrais de interesse deste grupo.

8.2 Trabalhos futuros

A seguir, algumas propostas para trabalhos futuros são apresentadas:

- **Completar o modelo.** Fases como a de testes e de gerência de processo ainda não foram devidamente endereçadas. Além do mais, não encontramos nenhuma referência sobre como tais aplicações possam ser efetivamente testadas e muito menos métricas de projeto voltadas para o desenvolvimento de aplicações baseadas em agentes móveis.
- **Validar o modelo.** Outros estudos de caso abordando diversos domínios de aplicações devem ser realizados e métricas devem ser coletadas a fim de avaliar o modelo e a qualidade dos seus artefatos.
- **Ferramentas de suporte ao desenvolvimento de aplicações baseadas em agentes móveis.** Para produzir a maioria dos artefatos de análise e projeto foi utilizada a ferramenta Rational Rose Enterprise Edition (http://www.rational.com). Entretanto, ferramentas adicionais são necessárias, particularmente suportando extensões de UML, técnicas de teste, engenharia reversa e geração de código e documentação.
- **Utilização de notações formais para expressar contratos e elementos arquiteturais.** Isto tornará possível, por exemplo, a geração automática de testes funcionais, bem como a análise mais precisa de propriedades de projeto.

9 APÊNDICE A - IMPLEMENTAÇÃO

A especificação lógica de projeto do Sistema de Apoio às Atividades de Comitês de Programa em Conferências consiste em um conjunto de classes de objetos e de classes de agentes. As classes de objetos foram implementadas em Java e testes de unidades foram realizados para cada uma delas, enquanto que as classes de agentes foram implementadas através do uso do *framework white box* fornecido pela API da plataforma Grasshopper.

Todas as classes, comportamentos e relacionamentos apresentados no projeto dependente de plataforma foram implementados conforme especificados no Capítulo 7, todavia, para manter a consistência, o projeto teve que ser alterado diversas vezes devido a detalhes específicos da plataforma que só foram identificados durante a implementação.

Algumas classes que não foram apresentadas nas especificações do projeto também foram implementadas, entre elas a classe *AplicaoConferencia* (ver Seção 9.1) e outras classes de suporte à persistência dos objetos.

As principais classes do sistema serão apresentadas nas próximas seções.

9.1 Classe AplicacaoConferencia

Esta classe consiste em uma aplicação stand-alone que interage com o ambiente Grasshopper como cliente do serviço de comunicação. É através da sua execução que o sistema é iniciado. Antes de executar a aplicação, a região da conferência (*RegiaoConferencia*) e agência do coordenador (*AgenciaCoordenador*) devem estar ativas.

No início do método `main()`, a *AplicaoConferencia* realiza os seguintes passos:

1. Cria o objeto *Conferencia*.
2. Cria o objeto de serviço de comunicação externo (*ExternalCommService*) e registra o objeto *Conferencia*.
3. Inicia o receptor de comunicação do objeto de serviço de comunicação externo.

Depois destes passos, o objeto *Conferencia* está acessível para clientes através do serviço de comunicação Grasshopper, desde que o cliente conheça o endereço de comunicação e a classe de interface do objeto *Conferencia*.

Agora, a *AplicacaoConferencia* age como cliente de diversos componentes Grasshopper realizando os seguintes passos:

4. Cria o proxy da *RegiaoConferencia*

5. Através do proxy da *RegiaoConferencia*, obtém uma lista de todas as agências registradas na região com o nome *AgenciaCoordenador*.
6. Cria o proxy da primeira agência encontrada.
7. Através do proxy da agência, cria o *AgenteCoordenador* na Agência do Coordenador, passando como parâmetro o endereço de comunicação do objeto *Conferencia*.

Código da classe *AplicacaoConferencia*:

```
Package agentes;

import de.ikv.grasshopper.communication.GrasshopperAddress;
import de.ikv.grasshopper.communication.ExternalCommService;
import de.ikv.grasshopper.communication.ServerStartUpException;
import de.ikv.grasshopper.communication.ProxyGenerator;
import de.ikv.grasshopper.agency.*;
import de.ikv.grasshopper.util.SearchFilter;
import de.ikv.grasshopper.type.AgentSystemInfo;
import de.ikv.grasshopper.type.AgentInfo;
import classes.Conferencia;
import classes.ServidorConferenciaIF;
import excecoes.*;

/**
 * Aplicação Conferência. Ao executar ela registra objeto externo Conferencia
 * no serviço de Comunicação Grasshopper e cria o AgenteCoordenador dentro da
 * AgenciaCoordenador.
 * @author  Fabiana Paulino Guedes
 * @version 1.0
 */
public class AplicacaoConferencia
{
  /**
   * Endereço da Região da Conferência onde a Agência do Coordenador está
   * registrada.
   */
  private static final String ENDREGIAO =
  "socket://localhost:7020/RegiaoConferencia";
  /**
   * Nome da Agência do Coordenador
   */
  private static final String NOMEAGENCIACOORDENADOR = "AgenciaCoordenador";
  /**
   * Code base do agente coordenador
   */
  private static final String CODEBASE =
  "http://www.dsc.ufpb.br/~patricia/fabiana/conferencia/";
  /**
   * O programa principal. Registra objeto externo Conferencia
   * no serviço de Comunicação Grasshopper e cria o AgenteCoordenador dentro
   * da AgenciaCoordenador.
   */
  public static void main(String args[])
  {
    // Endereço do serviço de comunicação
    GrasshopperAddress endServicoComm =
    new GrasshopperAddress("socket://localhost:7888/AgenciaCoordenador");
    try
    {
      // Cria o objeto servidor Conferência
      System.out.println("## AplicaçãoConferência: Criando o objeto " +
      "servidor Conferência");
      ServidorConferenciaIF conferencia = new Conferencia(1);
```

```
// Cria o serviço de comunicação externo e registra o objeto servidor
// Conferencia.
System.out.println("##AplicaçãoConferência: Criando o serviço de " +
"comunicação externo");
ExternalCommService servicoComm = new ExternalCommService(conferencia);
servicoComm.start();

// Inicia o receptor de comunicação
System.out.println("##AplicaçãoConferência: Iniciando novo receptor" +
" no " + endServicoComm);
servicoComm.startReceiver(endServicoComm);

// Contacta a região
System.out.println("##AplicaçãoConferência: Contactando a região '" +
ENDREGIAO + "'.");
GrasshopperAddress endGrassRegiao = new GrasshopperAddress(ENDREGIAO);
IRegionRegistration regionProxy = (IRegionRegistration)
ProxyGenerator.newInstance(IRegionRegistration.class,
endGrassRegiao.generateRegionId(), endGrassRegiao);

// Obtém uma lista de todas as agências registradas na região com o
// nome AgenciaCoordenador.
AgentSystemInfo[] agentSystemInfo = regionProxy.listAgencies(
new SearchFilter("NAME = " + NOMEAGENCIACOORDENADOR));
System.out.println("##AplicaçãoConferência: Agência do coordenador " +
                   "encontrada.");

if (agentSystemInfo.length > 0)
{
  // Contacta a primeira agência da lista
  System.out.println("## AplicaçãoConferência: Contactando agência '" +
                   AgentSystemInfo[0].getLocation().generateAgentSystemId()
                   + "'.");
  GrasshopperAddress endAgencia = agentSystemInfo[0].getLocation();
  String listaEndServidores[] = regionProxy.lookupCommunicationServer(
  endAgencia.generateAgentSystemId());
  System.out.println("## AplicaçãoConferência: Selecionando servidor " +
                     ListaEndServidores[0]);
  GrasshopperAddress agencyAddress =
                            new GrasshopperAddress(listaEndServidores[0]);
  IAgentSystem agencyProxy = (IAgentSystem)
  ProxyGenerator.newInstance(IAgentSystem.class,
  agencyAddress.generateAgentSystemId(), agencyAddress);

  // Cria o AgenteCoordenador na agência do coordenador.
  System.out.println("## AplicaçãoConferência: Criando agente na agência"
                     + "do coordenador");
  Object agentCreationArgs[] = new Object[1];
  agentCreationArgs[0] = (String)endServicoComm.toString();
  AgentInfo agentInfo = agencyProxy.createAgent(
           "agentes.AgenteCoordenador", CODEBASE, "", agentCreationArgs);
}
else
{
  System.out.println("## AplicaçãoConferência: Nada foi feito. " +
                     "Por favor inicie a agência do coordenador.");
}

try
{
  System.out.println("## AplicaçãoConferência: digite alguma coisa " +
                     "para sair.");
  int sair = System.in.read();
  System.out.println("## AplicaçãoConferência: Parando o serviço de " +
                     "comunicação.");
  servicoComm.stop();
  System.out.println("## AplicaçãoConferência: Pronto.");
```

```
        System.exit(0);
      }
      catch(java.io.IOException e)
      {
        System.out.println("## AplicaçãoConferência:" + e.getMessage());
      }
    }
    catch (AbrirConException e)
    {
      System.out.println("## AplicaçãoConferência: " + e.getMessage());
    }
    catch (GetDetalhesException e)
    {
      System.out.println("## AplicaçãoConferência: " + e.getMessage());
    }
    catch (GetIteradorException e)
    {
      System.out.println("## AplicaçãoConferência: " + e.getMessage());
    }
    catch (ServerStartUpException e)
    {
      System.out.println("## AplicaçãoConferência: " + e.getMessage());
    }
    catch (AgentSystemNotFoundException e)
    {
      System.out.println("## AplicaçãoConferência: " + e.getMessage());
    }
    catch (AgentCreationFailedException e)
    {
      System.out.println("## AplicaçãoConferência: " + e.getMessage());
    }
  }
}
```

9.2 Interface *ServidorConferenciaIF*

Esta interface representa a interface servidora que é implementada pela classe *Conferencia* e que é usada pelo *AgenteCoordenador* para criar um proxy do objeto servidor *Conferencia*.

A interface *ServidorConferenciaIF* define os seguintes métodos que serão acessados pelo *AgenteCoordenador* através de um proxy:

- `getColecaoDadosArtigos()`: obtém coleção de objetos da classe *DadosArtigo*.
- `getColecaoDadosMembroComite()`: obtém coleção de objetos da classe *DadosMembroComite*.
- `getDadosConferencia()`: obtém objeto com os dados da conferência.
- `criarRegistroRevisao(String idArtigo, String idMembroComite, String qntCopias)`: cria o registro de revisão de um artigo.

- `registrarResultado(DadosRegistroRevisaoIF dadosRegistroRevisao, DadosRevisaoIF dadosRevisao)`: registra o resultado da revisão do artigo.

Código da interface *ServidorConferenciaIF*:

```
package classes;

import java.io.*;
import java.util.Iterator;
import agentes.DadosConferenciaIF;
import detalhes.DetalhesConferenciaAbstractIF;
import excecoes.*;

import agentes.*;

/**
 * Interface do objeto servidor Conferencia
 * @author  Fabiana Paulino Guedes
 * @version 1.0
 */
public interface ServidorConferenciaIF extends DetalhesConferenciaAbstractIF
{

  /**
   * Obtém coleção de objetos DadosArtigo.
   * @return  Coleção de objetos DadosArtigo.
   */
  public ColecaoDadosArtigo getColecaoDadosArtigos()
         throws FileNotFoundException, IOException;

  /**
   * Obtém coleção de objetos DadosMembroComite.
   * @return  Coleção de objetos DadosMembroComite.
   */
  public ColecaoDadosMembroComite getColecaoDadosMembroComite();

  /**
   * Obtém objeto com os dados da conferência.
   * @return  Dados da Conferência.
   */
  public DadosConferenciaIF getDadosConferencia();

  /**
   * Cria o registro de revisão de um artigo.
   * @param <B>idArtigo</B> Identificador do artigo.
   * @param <B>idMembroComite</B> Identificador do Membro de comitê
   *        responsável pela revisão.
   * @param <B>qntCopias</B> Quantidade de cópias a serem entregues.
   * @return  Dados do registro de revisão criado.
   */
  public DadosRegistroRevisaoIF criarRegistroRevisao(String idArtigo,
         String idMembroComite, String qntCopias)
         throws AbrirConException, GetProximoIdException, InserirException,
                GetIteradorException, GetDetalhesException,
                FileNotFoundException, IOException;

  /**
   * Registra o resultado da revisão do artigo.
   * @param <B>dadosRegistroRevisao</B> Dados do registro de revisão
   * @param <B>dadosRevisão</B> Dados da Revisão
   */
  public void registrarResultado(DadosRegistroRevisaoIF dadosRegistroRevisao,
         DadosRevisaoIF dadosRevisao)
         throws AbrirConException, GetDetalhesException, GetProximoIdException,
                InserirException, GetIteradorException;
```

```
}
```

9.3 Classe *Conferencia*

Esta é a classe controladora da parte de objetos do sistema. Uma instância desta classe é registrada no serviço de comunicação Grasshopper atuando como servidora para o *AgenteCoordenador*.

Toda parte de persistência dos dados é tratada através de objetos. Sempre que o *AgenteCoordenador* precisa recuperar ou armazenar informações na base de dados, ele interage com o objeto *Conferencia* através de um proxy. O objeto *Conferencia* recebe as requisições e delega as tarefas para os objetos responsáveis.

Esta classe implementa os métodos definidos na interface *ServidorConferenciaIF*.

Código da classe *Conferencia*:

```
package classes;

import java.io.*;
import java.util.Iterator;
import java.util.Vector;
import java.util.Hashtable;
import excecoes.*;
import detalhes.DetalhesConferenciaAbstract;
import detalhes.DetalhesConferencia;
import detalhes.DetalhesConferenciaIF;

import agentes.*;

/**
 * Classe controladora do sistema.
 * Todo acesso aos objetos persistentes é realizado através de um objeto
 * desta classe.
 * @author  Fabiana Paulino Guedes
 * @version 1.0
 */
public class Conferencia extends DetalhesConferenciaAbstract
                         implements ServidorConferenciaIF
{

  /** Encapsula os métodos de persistência do objeto Conferência */
  private ConferenciaDBIF db;
  /** Coordenador da Conferência*/
  private CoordenadorIF coordenador;
  /** Comitê de Programa */
  private ComiteProgramaIF comite;
  /** Coleção de artigos */
  private ColecaoArtigosIF colArtigos;

  /**
   * Flag de controle. Identifica se os dados do Objeto conferência foram
   * alterados e precisam ser atualizados no bando de dados.
   */
  private boolean modificado = false;

  /**
   * Constrói um novo objeto Conferencia
```

```
 * @param <B>id</B> Identificador da Conferência
 * @param <B>coordenador</B> Coordenador da Conferência
 * @param <B>comite</B> Comitê de Programa
 * @param <B>titulo</B> Titulo da Conferência
 * @param <B>dtInicio</B> Data de Início da Conferência
 * @param <B>dtFinal</B> Data Final da Conferência
 * @param <B>enderco</B> Endereço da Conferência
 * @param <B>url</B> URL da Conferência
 */
public Conferencia(int id, CoordenadorIF coordenador,
       ComiteProgramaIF comite, String titulo, java.sql.Date dtInicio,
       java.sql.Date dtFinal, String endereco, String url)
       throws AbrirConException, GetIteradorException, GetDetalhesException
{
  db = criarConferenciaDB();
  setId(id);
  setCoordenador(coordenador);
  setComite(comite);
  setTitulo(titulo);
  setDtInicio(dtInicio);
  setDtFinal(dtFinal);
  setEndereco(endereco);
  setModificado(false);
  setUrl(url);
  colArtigos = criarColecaoArtigos();
}

/**
 * Constrói um novo objeto Conferencia
 * @param <B>coordenador</B> Coordenador da Conferência
 * @param <B>comite</B> Comitê de Programa
 * @param <B>titulo</B> Titulo da Conferência
 * @param <B>dtInicio</B> Data de Início da Conferência
 * @param <B>dtFinal</B> Data Final da Conferência
 * @param <B>enderco</B> Endereço da Conferência
 * @param <B>url</B> URL da Conferência
 */
public Conferencia(CoordenadorIF coordenador, ComiteProgramaIF comite,
       String titulo, java.sql.Date dtInicio, java.sql.Date dtFinal,
       String endereco, String url)
       throws AbrirConException, GetProximoIdException, InserirException,
              GetIteradorException, GetDetalhesException
{
  db = criarConferenciaDB();
  setId(db.getProximoId());
  setCoordenador(coordenador);
  setComite(comite);
  setTitulo(titulo);
  setDtInicio(dtInicio);
  setDtFinal(dtFinal);
  setEndereco(endereco);
  setUrl(url);
  db.inserir(this);
  setModificado(false);
  colArtigos = criarColecaoArtigos();
}

/**
 * Constrói um novo objeto Conferencia recuperando os dados que já existem
 * no B.D. através do indetificador.
 * @param <B>id</B> Identificador da Conferência
 */
public Conferencia(int id) throws AbrirConException, GetDetalhesException,
                                  GetIteradorException
{
  db = criarConferenciaDB();
  DetalhesConferenciaIF dt = db.getDetalhes(id);
  setId(dt.getId());
```

```
    setCoordenador(criarCoordenador(dt.getIdCoordenador()));
    setComite(criarComitePrograma(dt.getIdComite()));
    setTitulo(dt.getTitulo());
    setDtInicio(dt.getDtInicio());
    setDtFinal(dt.getDtFinal());
    setEndereco(dt.getEndereco());
    setUrl(dt.getUrl());
    setModificado(false);
    colArtigos = criarColecaoArtigos();
  }

  /**
   * Atribui um novo valor para o  identificador da Conferência.
   * @param <B>id</B> Identificador da Conferência.
   */
  public void setId(int id)
  {
    super.setId(id);
    setModificado(true);
  }

  /**
   * Atribui um novo valor para o  Coordenador da Conferência.
   * @param <B>coordenador</B> Coordenador da Conferência.
   */
  public void setCoordenador(CoordenadorIF coordenador)
  {
    this.coordenador = coordenador;
    setModificado(true);
  }

  /**
   * Obtém o Coordenador da Conferência.
   * @return Coordenador da Conferência.
   */
  public CoordenadorIF getCoordenador()
  {
    return coordenador;
  }

  /**
   * Atribui um novo valor para o  Comitê de Programa.
   * @param <B>comite</B> Comitê de Programa.
   */
  public void setComite(ComiteProgramaIF comite)
  {
    this.comite = comite;
    setModificado(true);
  }

  /**
   * Obtém o Comitê de Programa.
   * @return Comitê de Programa.
   */
  public ComiteProgramaIF getComite()
  {
    return comite;
  }

  /**
   * Atribui um novo valor para o título da Conferência.
   * @param <B>titulo</B> Título da Conferência.
   */
  public void setTitulo(String titulo)
  {
    super.setTitulo(titulo);
    setModificado(true);
  }
```

```
/**
 * Atribui um novo valor para a data de ínicio da Conferência.
 * @param <B>dtInicio</B> Data de ínicio da Conferência.
 */
public void setDtInicio(java.sql.Date dtInicio)
{
  super.setDtInicio(dtInicio);
  setModificado(true);
}

/**
 * Atribui um novo valor para a data final da Conferência.
 * @param <B>dtFinal</B> Data final da Conferência.
 */
public void setDtFinal(java.sql.Date dtFinal)
{
  super.setDtFinal(dtFinal);
  setModificado(true);
}

public void setEndereco(String endereco)
{
  super.setEndereco(endereco);
  setModificado(true);
}

/**
 * Atribui um novo valor para a URL Conferência.
 * @param <B>url</B> URL da Conferência.
 */
public void setUrl(String url)
{
  super.setUrl(url);
  setModificado(true);
}

/**
 * Atribui um novo valor para o flag modificado .
 * @param <B>modificado</B> Novo valor para o flag modificado.
 */
private void setModificado(boolean modificado)
{
  this.modificado = modificado;
}

/**
 * Obtém o valor do flag modificado .
 * @return true se algum dado do objeto Conferencia tiver sido alterado,
 *          false caso contrário.
 */
public boolean getModificado()
{
  return modificado;
}

/**
 * Atualiza os dados da Conferência no B.D.
 */
public void atualizarDB() throws AtualizarException
{
  db.atualizar(this);
  setModificado(false);
}

/**
 * Remove o registro da conferencia do B.D.
 */
```

```
public void removerDB() throws RemoverException
{
  db.remover(this.getId());
}

/**
 * Obtém um iterador com todos os objetos Conferencia que foram persistidos
 * no B.D.
 * @return Iterador de Conferencias
 */
public static Iterator getIteradorDB() throws AbrirConException,
                                              GetIteradorException
{
  ConferenciaDBIF dbLocal = criarConferenciaDB();
  return dbLocal.getIterador();
}

/**
 * Obtém iterador de artigos
 * @return iterador de artigos
 */
public Iterator getIteradorArtigos()
{
  return colArtigos.getIterador();
}

/**
 * Obtém coleção de objetos DadosArtigo.
 * @return  Coleção de objetos DadosArtigo.
 */
public ColecaoDadosArtigo getColecaoDadosArtigos() throws FileNotFoundException,
                                                          IOException
{
  Iterator itAux = colArtigos.getIterador();
  ColecaoDadosArtigo colDadosArtigo = new ColecaoDadosArtigo();
  while(itAux.hasNext())
  {
    ArtigoIF artigo = (ArtigoIF) itAux.next();
    DadosArtigoIF dadosArtigo = criarDadosArtigo(artigo.getId(),
    artigo.getTitulo(), artigo.getArquivo());
    colDadosArtigo.inserir(dadosArtigo.getId(), dadosArtigo);
  }
  return colDadosArtigo;
}

/**
 * Obtém coleção de objetos DadosMembroComite.
 * @return  Coleção de objetos DadosMembroComite.
 */
public ColecaoDadosMembroComite getColecaoDadosMembroComite()
{
  Iterator itAux = comite.getIteradorMembrosComite();
  ColecaoDadosMembroComite colDadosMembroComite =
    new ColecaoDadosMembroComite();
  while(itAux.hasNext())
  {
    MembroComiteIF membro = (MembroComiteIF) itAux.next();
    DadosMembroComiteIF dadosMembroComite =
      criarDadosMembroComite(membro.getId(), membro.getNome(),
                             membro.getEndRegiao(), membro.getNomeAgencia());
    colDadosMembroComite.inserir(membro.getId(), dadosMembroComite);
  }
  return colDadosMembroComite;
}

/**
 * Obtém objeto com os dados da conferência.
 * @return  Dados da Conferência.
```

```
 */
public DadosConferenciaIF getDadosConferencia()
{
  return this.criarDadosConferencia(getTitulo(), getDtInicio(), getDtFinal(),
                                    getEndereco(), getUrl());
}

/**
 * Cria o registro de revisão de um artigo.
 * @param <B>idArtigo</B> Identificador do artigo.
 * @param <B>idMembroComite</B> Identificador do Membro de comitê
 *        responsável pela revisão.
 * @param <B>qntCopias</B> Quantidade de cópias a serem entregues.
 * @return  Dados do registro de revisão criado.
 */
public DadosRegistroRevisaoIF criarRegistroRevisao(String idArtigo,
       String idMembroComite, String qntCopias)
       throws AbrirConException, GetProximoIdException, InserirException,
              GetIteradorException, GetDetalhesException,
              FileNotFoundException, IOException
{
  System.out.println("IdArtigo: " + idArtigo);
  System.out.println("IdMembroComite: " + idMembroComite);
  System.out.println("QuantCopias: " + qntCopias);
  ArtigoIF artigo = colArtigos.getElemento(Integer.parseInt(idArtigo));
  MembroComiteIF membroComite =
    comite.getMembroComite(Integer.parseInt(idMembroComite));
  int idRegistro =
    artigo.inserirRegistroRevisao(membroComite, Integer.parseInt(qntCopias));
  RegistroRevisaoIF registro = artigo.getRegistroRevisao(idRegistro);


  DadosArtigoIF dadosArtigo =
    criarDadosArtigo(artigo.getId(), artigo.getTitulo(), artigo.getArquivo());
  DadosMembroComiteIF dadosMembroComite =
    criarDadosMembroComite(membroComite.getId(), membroComite.getNome(),
                           membroComite.getEndRegiao(),
                           membroComite.getNomeAgencia());
  DadosRegistroRevisaoIF dadosRegistroRevisao =
    criarDadosRegistroRevisao(registro.getId(), dadosArtigo,
                              dadosMembroComite, registro.getDtEnvio(),
                              registro.getQntCopias());
  return dadosRegistroRevisao;
}


/**
 * Registra o resultado da revisão do artigo.
 * @param <B>dadosRegistroRevisao</B> Dados do registro de revisão
 * @param <B>dadosRevisão</B> Dados da Revisão
 */
public void registrarResultado(DadosRegistroRevisaoIF dadosRegistroRevisao,
DadosRevisaoIF dadosRevisao)
throws AbrirConException, GetDetalhesException,
GetProximoIdException, InserirException,
GetIteradorException
{
  int idArtigo = dadosRegistroRevisao.getDadosArtigo().getId();
  int idRegistro = dadosRegistroRevisao.getId();
  DadosRevisorIF dadosRevisor = dadosRevisao.getDadosRevisor();
  int originalidade = Integer.parseInt(dadosRevisao.getOriginalidade());
  int legibilidade = Integer.parseInt(dadosRevisao.getLegibilidade());
  int relevancia = Integer.parseInt(dadosRevisao.getRelevancia());
  int aceitacao = Integer.parseInt(dadosRevisao.getAceitacao());
  String comentarios = dadosRevisao.getComentarios();
  String comentariosSigilosos = dadosRevisao.getComentariosSigilosos();

  ArtigoIF artigo = colArtigos.getElemento(idArtigo);
```

```
    RegistroRevisaoIF registroRevisao = artigo.getRegistroRevisao(idRegistro);
    Revisor revisor =
      new Revisor(dadosRevisor.getNome(), dadosRevisor.getInstituicao(),
                  dadosRevisor.getEndereco(), dadosRevisor.getFone(),
                  dadosRevisor.getFax(), dadosRevisor.getEmail(),
                  dadosRevisor.getDestino().getEndRegiao(),
                  dadosRevisor.getDestino().getNomeAgencia());
    registroRevisao.inserirFormRevisao(revisor, originalidade, legibilidade,
                                       relevancia, aceitacao, comentarios,
                                       comentariosSigilosos);
  }

  /**
   * Factory method. Cria um objeto do tipo ConferenciaDBIF
   * @return Objeto do tipo ConferenciaDBIF
   */
  private static ConferenciaDBIF criarConferenciaDB() throws AbrirConException
  {
    return ConferenciaDB.getInstance();
  }

  /**
   * Factory method. Cria um objeto do tipo CoordenadorIF recuperando os dados
   * no B.D. a partir do identificador.
   * @param <B>id</B> Identificador do coordenador
   * @return Objeto do tipo CoordenadorIF
   */
  private CoordenadorIF criarCoordenador(int idCoordenador)
          throws AbrirConException, GetDetalhesException
  {
    return new Coordenador(idCoordenador);
  }

  /**
   * Factory method. Cria um objeto do tipo ComiteProgramaIF recuperando os dados
   * no B.D. a partir do identificador.
   * @param <B>id</B> Identificador do comitê de programa
   * @return Objeto do tipo ComiteProgramaIF
   */
  private ComiteProgramaIF criarComitePrograma(int idComite)
          throws AbrirConException, GetDetalhesException, GetIteradorException
  {
    return new ComitePrograma(idComite);
  }

  /**
   * Factory method. Cria um objeto do tipo ColecaoArtigosIF
   * @return Objeto do tipo ColecaoArtigosIF
   */
  private ColecaoArtigosIF criarColecaoArtigos()
          throws AbrirConException, GetIteradorException, GetDetalhesException
  {
    return new ColecaoArtigos();
  }

  /**
   * Factory method. Cria um objeto do tipo DadosConferenciaIF
   * @param <B>titulo</B> Titulo da Conferência
   * @param <B>dtInicio</B> Data de Início da Conferência
   * @param <B>dtFinal</B> Data Final da Conferência
   * @param <B>enderco</B> Endereço da Conferência
   * @param <B>url</B> URL da Conferência
   * @return Objeto do tipo DadosConferenciaIF
   */
  private DadosConferenciaIF criarDadosConferencia(String titulo, java.sql.Date
  dtInicio, java.sql.Date dtFinal,
  String endereco, String url)
  {
```

```
    return new DadosConferencia(titulo, dtInicio, dtFinal, endereco, url);
  }

  /**
   * Factory method. Cria um objeto do tipo DadosArtigoIF
   * @param <B>id</B> Identificador do artigo
   * @param <B>titulo</B> Título do artigo
   * @param <B>arquivo</B> Array de bytes com o conteúdo do arquivo do artigo.
   * @return Objeto do tipo DadosArtigoIF
   */
  private DadosArtigoIF criarDadosArtigo(int id, String titulo, byte[] arquivo)
  {
    return new DadosArtigo(id, titulo, arquivo);
  }

  /**
   * Factory method. Cria um objeto do tipo DadosMembroComiteIF
   * @param <B>id</B> Identificador do membro de comitê
   * @param <B>nome</B> Nome do membro de comitê
   * @param <B>endRegiao</B> Endereço da região onde a agência do membro de
   *        comitê está registrada.
   *  @param <B>nome</B> Nome da agência do membro de comitê.
   * @return Objeto do tipo DadosMembroComiteIF
   */
  private DadosMembroComiteIF criarDadosMembroComite(int id, String nome,
          String endRegiao, String nomeAgencia)
  {
    return new  DadosMembroComite(id, nome, endRegiao, nomeAgencia);
  }

  /**
   * Factory method. Cria um objeto do tipo DadosRegistroRevisaoIF
   * @param <B>id</B> Identificador do registro de revisão
   * @param <B>dadosArtigo</B> Dados do Artigo
   * @param <B>dadosMembroComite</B> Dados do membro de comitê
   * @param <B>dtEnvio</B> Data de Envio
   * @param <B>qntCopias</B> Quantidade de Cópias
   * @return Objeto do tipo DadosRegistroRevisaoIF
   */
  private DadosRegistroRevisaoIF criarDadosRegistroRevisao(int id,
          DadosArtigoIF dadosArtigo, DadosMembroComiteIF dadosMembroComite,
          java.sql.Date dtEnvio, int qntCopias)
  {
    return new  DadosRegistroRevisao(id, dadosArtigo, dadosMembroComite, dtEnvio,
                                    qntCopias);
  }
}
```

9.4 Interface *AgenteCoordenadorIF*

Interface que define a operação do sistema `gerarFormRevisao(...)`. Esta interface é implementada pela classe *AgenteCoordenador* e é utilizada pela interface gráfica deste para criar o proxy dele e assim poder repassar as solicitações do usuário.

```
package agentes;

/**
 * Interface do Agente Coordenador com os métodos disponíveis para a GUI.
 * @author  Fabiana Paulino Guedes
 * @version 1.0
```

```
 */
public interface AgenteCoordenadorIF
{

  /**
   * Gera o AgenteFormularioRevisao. Este método é invocado pela GUI do agente.
   * @param <B>idArtigo</B> Identificador do artigo a ser revisado
   * @param <B>idMembroComite </B> Identificador do membro de comitê responsável
   *         pelo redirecionamento do formuário de revisão.
   * @param <B>qntCopias</B> Quantidade de cópias a serem entregues para
   *        o membro do comitê.
   */
  public void gerarFormRevisao(String idArtigo, String idMembroComite,
         String qntCopias);

}
```

9.5 Interface *AgenteMestreIF*

Esta interface é implementada pelo *AgenteCoordenador* (agente mestre) e define o método `registrarResultado(...)` que será acessado através de um proxy pelo *AgenteFormRevisao* (agente escravo).

Código da interface *AgenteMestreIF*:

```
package agentes;

import excecoes.AbrirConException;
import excecoes.GetProximoIdException;
import excecoes.InserirException;
import excecoes.GetIteradorException;
import excecoes.GetDetalhesException;

/**
 * Interface com os métodos disponíveis pelo AgenteCoordenador (agente mestre)
 * para o AgenteFormRevisao (agente escravo).
 * @author  Fabiana Paulino Guedes
 * @version 1.0
 */
public interface AgenteMestreIF
{

  /**
   * Registra os resultado da revisão do artigo.
   * Este método é acessado pelo Agente Formulário de Revisão quando ele
   * retorna para a agência do Coordrmador de Programa quando o processo de
   * revisão é finalizado.
   * @param <B>dadosRegistroRevisao</B> Dados do registro de revisão
   * @param <B>dadosRevisao</B> Dados da revisão
   */
  public void registrarResultado(DadosRegistroRevisaoIF dadosRegistroRevisao,
         DadosRevisaoIF dadosRevisao)
         throws AbrirConException, GetDetalhesException, GetProximoIdException,
                InserirException, GetIteradorException;
}
```

9.6 Classe *AgenteCoordenador*

Classe que representa o agente coordenador. O agente coordenador age em nome do coordenador de programa e é o responsável por gerenciar todas as atividades do Sistema de Apoio às Atividades de um Comitê de Programa.

O *AgenteCoordenador* é um agente estacionário e persistente que estende a classe `de.ikv.grasshopper.agent.SmartServer` da API Grasshopper e implementa as interfaces *AgenteCoordenadorIF* e *AgenteMestreIF*.

O *AgenteCoordenador* mantém as seguintes variáveis de instância não transientes, ou seja, que permanecem no estado de dados do agente:

- `endServidorConferencia:` endereço de comunicação do objeto externo servidor *Conferencia*. Este endereço é entregue ao agente através do argumento de criação.
- `proxyConferencia`: um proxy do objeto servidor *Conferencia*, criado dentro do método `init(...)` do agente.
- `waitLock:` objeto serializável usado para sincronizar a notificação de espera.

Além destas o *AgenteCoordenador* mantém a variável de instância transiente `gui`, `componente de interface gráfica do agente.`

O método `live()` define o comportamento ativo do agente, ou seja, controla o fluxo que é realizado dentro da própria thread do agente. O método `live()` é declarado abstrato na superclasse `de.ikv.grasshopper.agency.Agent` e deve ser obrigatoriamente sobrescrito por cada agente Grasshopper. O comportamento ativo do *AgenteCoordenador* resume-se em:

- criar um proxy de si mesmo para passar como argumento de criação do componente de interface gráfica que é criado logo em seguida;
- criar o componente de interface gráfica;
- exibir a G.U.I. (ver Figura 9.1) e esperar por alguma solicitação do usuário.

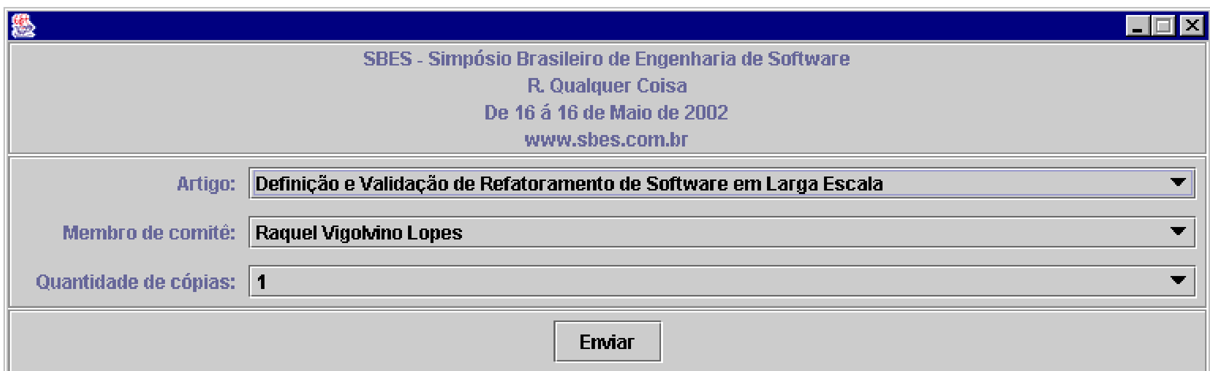

Figura 9.1: G.U.I. do Agente Coordenador

Quando o usuário clica no botão enviar, a G.U.I. invoca (através de um proxy) o método `gerarFormRevisao(...)` do *AgenteCoordenador* que está definido na interface *AgenteCoordenadorIF*. O *AgenteCoordenador* solicita ao objeto servidor *Conferencia* (através do proxy) a criação do registro de revisão para o artigo selecionado e solicita à agência em que está hospedado (através de um proxy) a criação do *AgenteFormRevisao*, passando como argumentos de criação os dados da conferencia e os dados do registro de revisão recém criado. Os dados do registro de revisão contém dados sobre o artigo, o membro de comitê e a quantidade de cópias selecionados.

O método `registrarResultado(...)` está definido na interface *AgenteMestreIF* e é invocado pelo *AgenteFormRevisao* quando ele retorna a agência do coordenador após o término da revisão do artigo. Quando este método é invocado, o *AgenteCoordenador* delega a ao objeto servidor *Conferencia* a responsabilidade de registrar o resultado da revisão.

Código da classe *AgenteCoordenador*:

```
package agentes;

import de.ikv.grasshopper.agent.SmartServer;
import de.ikv.grasshopper.agency.IAgentSystem;
import de.ikv.grasshopper.agency.AgentCreationFailedException;
import de.ikv.grasshopper.type.AgentInfo;
import de.ikv.grasshopper.communication.ProxyGenerator;
import de.ikv.grasshopper.communication.GrasshopperAddress;

import classes.ServidorConferenciaIF;
import excecoes.AbrirConException;
import excecoes.GetProximoIdException;
import excecoes.InserirException;
import excecoes.GetIteradorException;
import excecoes.GetDetalhesException;


/**
 * Classe do Agente Coordenador que é um agente do tipo estacionário
 * persistente. Este agente age em nome do coordenador de programa e é o
 * responsável por gerenciar todas as atividades do Sistema de Apoio às
 * Atividades de um Comitê de Programa.
 * @author  Fabiana Paulino Guedes
 * @version 1.0
 */
public class AgenteCoordenador extends SmartServer
       implements AgenteCoordenadorIF, AgenteMestreIF
{

  /**
   * Endereço de comunicação do objeto externo servidor conferência.
   * Faz parte do estado de dados do agente, desde que é não transiente.
   */
  private GrasshopperAddress endServidorConferencia;
  /**
   * Proxy do objeto servidor conferência.
   * Faz parte do estado de dados do agente, desde que é não transiente.
   */
  private ServidorConferenciaIF proxyConferencia;
```

```
/**
 * Usado para sincronizar a noticação de espera.
 * Obs.: O objeto precisa ser serializável
 */
protected Object waitLock = new String();
/**
 * Interface gráfica do Agente Coordenador.
 */
transient private GuiAgenteCoordenador gui;


/**
 * Obtém o nome do agente
 * @return nome do agente
 */
public String getName()
{
  return "AgenteCoordenador";
}

/**
 * Inicializa o agente.
 * Argumentos de criação: arg[0] = endereço de comunicação do objeto externo
 * ServidorConferencia.
 * @param <B>args</B> Vetor com argumentos de criação.
 */
public void init(Object[] args)
{
  if (args.length < 1)
  {
    log("Está faltando passar o argumento de criação: <endServidorConferencia>");
    log("Saindo.");
    throw new RuntimeException("Está faltando argumentos");
  }

  //Pega o endereço do objeto externo ServidorConferencia
  String endSrvdConferenciaStr = (String)args[0];
  // Contacta o objeto ServidorConferencia que foi criado pela
  //aplicação servidora
  log("Criando proxy para o objeto externo conferencia...");
  endServidorConferencia = new GrasshopperAddress(endSrvdConferenciaStr);
  proxyConferencia =
    (ServidorConferenciaIF) ProxyGenerator.newInstance(
      classes.ServidorConferenciaIF.class, "", endServidorConferencia);
  log("Proxy do objeto ServidorConferencia criado com sucesso.");
}

/**
 * Contém o comportamento do agente. O agente exibe uma GUI e fica aguardando
 * as solicitações do usúario.
 */
public void live()
{
  try
  {

    // Cria um proxy de si mesmo para passar como argumento de criação da GUI
    AgenteCoordenadorIF proxyAgenteCoordenador =
    (AgenteCoordenadorIF) getAgentProxy(this.getInfo(),
    AgenteCoordenadorIF.class);
    // Cria a GUI e a exibe
    gui = new GuiAgenteCoordenador(proxyAgenteCoordenador,
    proxyConferencia.getDadosConferencia(),
    proxyConferencia.getColecaoDadosArtigos(),
    proxyConferencia.getColecaoDadosMembroComite());
    gui.show();

    while (true)
```

```
    {
      //Espera por uma solicitação do usuário
      waitForEvent();
    }
  }
  catch(Exception e)
  {
    log(e.getMessage());
  }
}

/**
 * Este método é chamado quando o agente e invocado pelo usuário através da
 * agência.
 */
public void action()
{
  gui.show();
}

/**
 * Gera o AgenteFormularioRevisao. Este método é invocado pela GUI do agente.
 * @param <B>idArtigo</B> Identificador do artigo a ser revisado
 * @param <B>idMembroComite </B> Identificador do membro de comitê responsável
 *         pelo redirecionamento do formuário de revisão.
 * @param <B>qntCopias</B> Quantidade de cópias a serem entregues para
 * o membro do comitê.
 */
public void gerarFormRevisao(String idArtigo, String idMembroComite,
       String qntCopias)
{

  log("Gerando o agente formulário de revisão...");
  try
  {
    //Pega os dados da conferência.
    DadosConferenciaIF dadosConferencia =
      proxyConferencia.getDadosConferencia();
    //Cria um registro de revisão de artigo.
    DadosRegistroRevisaoIF dadosRegistro =
      proxyConferencia.criarRegistroRevisao(idArtigo, idMembroComite, qntCopias);
    //Cria o agente formulário de revisão.
    CriarAgenteFormRevisao(dadosConferencia, dadosRegistro);
    //Força o retorno do método.
    notifyEvent();
  } catch (AbrirConException e)
  {
    log(e.getMessage());
  } catch (GetProximoIdException e)
  {
    log(e.getMessage());
  } catch (InserirException e)
  {
    log(e.getMessage());
  } catch (GetIteradorException e)
  {
    log(e.getMessage());
  } catch (GetDetalhesException e)
  {
    log(e.getMessage());
  } catch (AgentCreationFailedException e)
  {
    log("Erro ao criar o AgenteFormRevisao: " + e.getMessage());
  } catch (Exception e)
  {
    log(e.getMessage());
  } catch (Throwable t)
  {
```

```
      log("Erro", t);
    }
    log("O AgenteFormRevisao foi criado com sucesso!!!");

  }

  /**
   * Cria o agente formulário de revisão a partir dos dados do regsitro de
   * revisão passados como parâmetro.
   * @param <B>dadosConferencia</B> Dados sobre a conferência.
   * @param <B>dadosRegistroRevisao</B> Dados do registro de revisão (artigo,
   * (membro de comitê, quantidade de cópias)
   */
  private void criarAgenteFormRevisao(DadosConferenciaIF dadosConferencia,
          DadosRegistroRevisaoIF dadosRegistroRevisao)
          throws AgentCreationFailedException
  {
    //Cria um proxy da agência hospedeira
    IAgentSystem proxyAgencia = this.getAgentSystem();
    //Array com argumetos de criação do agente formulário de revisão
    Object agentCreationArgs[] = new Object[2];
    agentCreationArgs[0] = dadosConferencia;
    agentCreationArgs[1] = dadosRegistroRevisao;
    //Solicita a agência hospedeira que crie o agente formulário de revisão.
    AgentInfo info = proxyAgencia.createAgent("agentes.AgenteFormRevisao",
    getInfo().getCodebase(),
    getInfo().getLocation().getPlace(), agentCreationArgs);
  }

  /**
   * Registra os resultado da revisão do artigo.
   * Este método é acessado pelo Agente Formulário de Revisão quando ele
   * retorna para a agência do Coordrmador de Programa quando o processo de
   * revisão é finalizado.
   * @param <B>dadosRegistroRevisao</B> Dados do registro de revisão
   * @param <B>dadosRevisao</B> Dados da revisão
   */
  public synchronized void  registrarResultado(
  DadosRegistroRevisaoIF dadosRegistroRevisao, DadosRevisaoIF dadosRevisao)
  throws AbrirConException, GetDetalhesException,
  GetProximoIdException, InserirException, GetIteradorException
  {
    log("Registrando o resultado da revisão... ");
    if (dadosRegistroRevisao != null && dadosRevisao != null)
    {
      log("Identificador do artigo: " +
dadosRegistroRevisao.getDadosArtigo().getId());
      log("Título do artigo: " +
dadosRegistroRevisao.getDadosArtigo().getTitulo());
      log("Identificador do Membro de Comitê: " +
dadosRegistroRevisao.getDadosMembroComite().getId());

      log("Nome Revisor: " + dadosRevisao.getDadosRevisor().getNome());
      log("Originalidade: " + dadosRevisao.getOriginalidade());
      log("Legibilidade: " + dadosRevisao.getLegibilidade());
      log("Relevância: " + dadosRevisao.getRelevancia());
      log("Aceitação: " + dadosRevisao.getAceitacao());
      log("Comentários: " + dadosRevisao.getComentarios());
      log("Comentários Sigilosos: " + dadosRevisao.getComentariosSigilosos());
      //Solicita ao objeto externo ServidorConferencia que regsitre os dados da
revisão.
      proxyConferencia.registrarResultado(dadosRegistroRevisao, dadosRevisao);
    }
    else
      log("Não foi possível registrar o resultado da revisão.");
    log("Fim de registrar resultado.");
    //Força o retorno do método.
    notifyEvent();
```

```
  }

  /**
   * Espera até que alguma coisa especial aconteça.
   * Use este método para bloquear o método life() até que alguma coisa
   * aconteça.
   */
  protected void waitForEvent()
  {
    waitForEvent(-1);
  }

  /**
   * Espera durante um certo tempo até que algum evento seja notificado.
   * Use este método para bloquear o método life() durante um certo tempo
   * até que algum evento seja notificado. Se timeout igual a -1 espera pela
   * notificação de um evento indefinidamente.
   * @param <B>timeout</B> Tempo em  milesegundos.
   */
  protected void waitForEvent(long timeout)
  {

    synchronized(waitLock)
    {
      try
      {
        if (timeout == -1)
        {
          waitLock.wait();
        }
        else
        {
          waitLock.wait(timeout);
        }
      }
      catch(InterruptedException e)
      {
        log("interrupted");
      }
    }
  }


  /**
   * Notifica que ocorreu um evento.
   * Chame este método de qualquer outra thread para notificar o agente.
   */
  protected void notifyEvent()
  {
    synchronized(waitLock)
    {
      waitLock.notify();
    }
  }
}
```

9.7 Interface *AgenteFormRevisaoIF*

Esta interface é implementada pela classe *AgenteFormRevisao* e é utilizada pela interface gráfica deste para criar o proxy dele e assim poder repassar as solicitações do usuário.

Código da interface *AgenteFormRevisaoIF*:

```
package agentes;

import excecoes.MigrarException;

/**
 * Interface do AgenteFormRevisao com os métodos disponíveis para a GUI.
 * @author  Fabiana Paulino Guedes
 * @version 1.0
 */
public interface AgenteFormRevisaoIF
{

  /**
   * Redireciona o agente para uma outra agência.
   * @param <B>endRegiao</B> Endereço da região na qual a agência destino está
   * registrada. Formato: protocolo://nomeMaquina:numeroPorta/nomeRegiao. Ex.:
   * socket://10.6.40.32:7020/RegiaoConferencia.
   * @param <B>nomeAgencia</B> Nome da agência destino
   */
  public void redirecionar(String endRegiao, String nomeAgencia, boolean retornar)
         throws MigrarException;

  /**
   * Atribui os dados do revisor e os dados da revisão.
   * @param <B>nomeRevisor</B> Nome do revisor
   * @param <B>instituicaoRevisor</B> Instituição do revisor
   * @param <B>endRevisor</B> Endereço do revisor
   * @param <B>foneRevisor</B> Telefone de contato do revisor
   * @param <B>faxRevisor</B> Fax do revisor
   * @param <B>emailRevisor</B> Endereço eletrônico do revisor
   * @param <B>originalidade</B> Nível de originalidade do artigo.
   * @param <B>legibilidade</B> Nível de legibilidade do artigo.
   * @param <B>relevancia</B> Nível de relevância do artigo.
   * @param <B>aceitacao</B> Nível de aceitação  do artigo.
   * @param <B>comentarios</B> Comentários adicionais.
   * @param <B>comentariosSigilosos</B> Comentários sigilosos.
   */
  public void entrarDadosRevisao(String nomeRevisor, String instituicaoRevisor,
  String endRevisor, String foneRevisor,
  String faxRevisor, String emailRevisor,
  String originalidade, String legibilidade,
  String relevancia, String aceitacao,
  String comentarios, String comentariosSigilosos);
  /**
   * Finaliza o processo de revisão mudando o estado do agente para em aprovação.
   */
  public void finalizarRevisao();

  /**
   * Aprova a revisão.
   */
  public void aprovarRevisao();

  /**
   * Obtém os dados da Conferência.
   * @return Dados da Conferência.
   */
  public DadosConferenciaIF getDadosConferencia();

  /**
   * Obtém os dados do registro de revisão do artigo.
   * @return Dados do registro de revisão do artigo.
   */
  public DadosRegistroRevisaoIF getDadosRegistroRevisao();

  /**
```

```
   * Obtém os dados da revisão do artigo.
   * @return Dados da revisão do artigo.
   */
  public DadosRevisaoIF getDadosRevisao();

  /**
   * Obtém estado do agente.
   * @return Estado do agente.
   */
  public int getEstado();

  /**
   * Extrai o arquivo do artigo a ser revisado em um determinado caminho.
   * @param <B>caminho</B> diretorio/nomeArquivo.pdf
   */
  public void extrairArquivo(String dir);
}
```

9.8 Classe *AgenteFormRevisao*

Classe que representa o agente formulário de revisão. Este agente é responsável por transportar os dados da revisão entre as máquinas do coordenador de programa, do membro de comitê e dos revisores.

O *AgenteFormRevisao* é um agente móvel e persistente que estende a classe `de.ikv.grasshopper.agent.SmartAgent` da API Grasshopper. Este agente implementa uma variação do padrão Mestre-Escravo Itinerante onde o *AgenteCoordenador* assume o papel de agente mestre e o *AgenteFormRevisao* o papel de agente escravo itinerante.

O *AgenteFormRevisao* mantém as seguintes variáveis de instância não transientes, ou seja, que permanecem no estado de dados do agente:

- `dadosConferencia: objeto com os dados sobre a conferência`. Este objeto é entregue ao *AgenteFormRevisao* pelo *AgenteCoordenador* através do argumento de criação.
- `dadosRegistroRevisao`: objeto com os dados sobre o registro de revisão do artigo. Este objeto é entregue ao *AgenteFormRevisao* pelo *AgenteCoordenador* através do argumento de criação.
- `dadosRevisao:` dados sobre a revisão.
- `itinerario:` objeto com o itinerário do agente. Este objeto é quem conhece o próximo destino do agente.
- `endAgenciaCoordenador`: endereço da agência do coordenador de programa, guardado para ser utilizado no retorno do agente. Todo agente Grasshopper possui a propriedade `home` que mantém o endereço da agência onde o agente foi criado, esta informação é suficiente para que o *AgenteFormRevisao* original saiba voltar para agência do coordenador de programa, visto que ele foi criado lá, mas seus clones serão criados na agência do membro de comitê e o endereço mantido pela propriedade `home` será correspondente ao da agência onde foram criados. A solução encontrada foi a criação do atributo `endAgenciaCoordenador`, que é inicializado no método `init(...)` com o endereço mantido no home do agente, que corresponde ao endereço da agência do coordenador. O método `init(...)` só é executado uma vez logo após a criação do

agente, desta forma o conteúdo do atributo `endAgenciaCoordenador` dos clones gerados será exatamente igual ao do agente original.

- `estado`: mantém o estado atual do agente (CRIADOAGORA, EMCLONAGEM, EMDISTRIBUICAO, EMREVISAO, EMAPROVACAO, RETORNOU). É através do valor deste atributo que o agente sabe que ação deve executar após cada migração.
- `estadoAnterior`: mantém o estado anterior do agente, para ser utilizado no caso de ocorrer alguma exceção durante a execução da tarefa e fazer com que o agente volte ao estado anterior.
- `waitLock`: objeto serializável usado para sincronizar a notificação de espera.

As seguintes constantes foram definidas na classe *AgenteFormRevisao* para serem atribuídas ao atributo `estado` do agente:

- `CRIADOAGORA`: estado assumido pelo agente logo após a sua criação. Valor igual a 0.
- `EMCLONAGEM`: estado assumido pelo agente antes de migrar da agência do coordenador de programa para a agência do membro de comitê onde ele deverá se clonar. Valor igual 1.
- `EMDISTRIBUICAO`: estado assumido pelo agente antes de ser clonado. Indica que está aguardo ser redirecionado pelo membro de comitê para um revisor. Valor igual a 2.
- `EMREVISAO`: estado assumido pelo agente antes ser redirecionado pelo membro de comitê para um revisor. Indica que o processo de revisão já foi iniciado, mas que ainda está em aberto. Valor igual a 3.
- `EMAPROVACAO`: estado assumido pelo agente após ter sido revisado. Valor igual a 4.
- `RETORNOU`: estado assumido pelo agente após ter retornado para agência do coordenador de programa. Valor igual a 5.

O *AgenteFormRevisao* também possui a variável de instância transiente `gui`, componente de interface gráfica do agente.

O método `init()` do *AgenteFormRevisao*, que é chamado pela agência logo após a sua criação, obtém os objetos *dadosConferencia* e *dadosRegistroRevisao* através dos parâmetros de criação e invoca o método `inicializarTarefa()`. Observe que antes de atribuirmos os parâmetros de criação aos atributos tivemos que reconstruir os objetos com seus tipos específicos através de reflexão. Segundo a documentação Grasshopper isso não seria necessário bastando apenas fazer um *cast* do parâmetro, que é do tipo object, para o tipo do atributo e efetuar a atribuição. Inicialmente tínhamos feito isto, mas uma exceção era lançada informando que não era possível realizar o *cast*, desta forma resolvemos reconstruir os objetos por meio de reflexão.

O comportamento ativo do *AgenteFormRevisao* segue uma adaptação do padrão Mestre-Escravo Itinerante que consiste em executar uma tarefa (que dependerá do valor do estado do agente) e migrar para o próximo destino. Em cada destino o agente executa novamente os mesmos passos até voltar para a agência do coordenador de programa quando o seu estado será igual a RETORNOU e ele entregará o resultado da revisão para o *AgenteCoordenador* e se destruirá.

O método `executarTarefa()` invocado dentro do método `live()` do *AgenteFormRevisao* é quem de fato controla o fluxo de execução das tarefas do agente. O que determina o que o agente fará é o valor do atributo `estado`. Se o estado do *AgenteFormRevisao* for igual a:

- `CRIADOAGORA`: o agente muda o valor do seu `estado` para `EMCLONAGEM`
- `EMCLONAGEM`: antes de se clonar o agente muda o seu `estado` para em distribuição (a fim de evitar que os clones também gerem clones de si próprios e assim sucessivamente) e gera a quantidade de cópias solicitadas.
- `EMDISTRIBUICAO`: o agente exibe sua G.U.I. (ver Figura 9.2) e aguarda que o membro de comitê o redirecione para algum revisor.

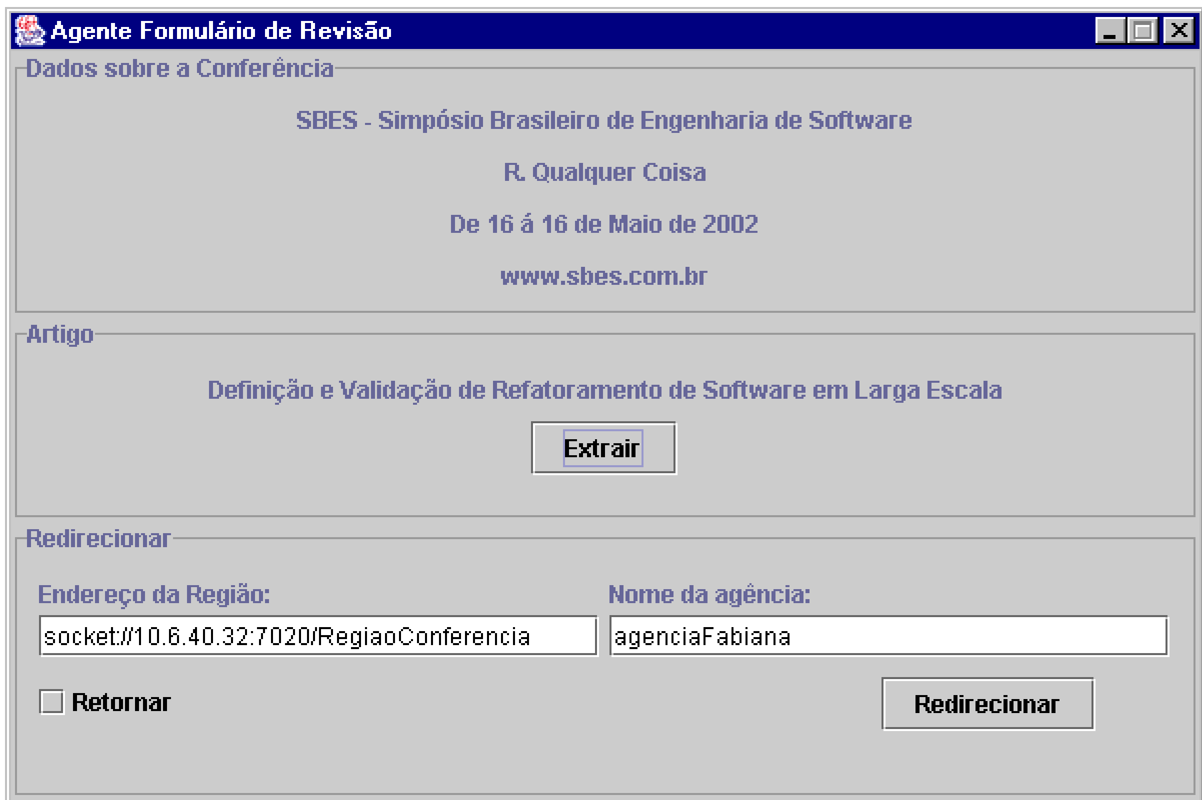

Figura 9.2: G.U.I. do AgenteFormRevisao em distribuição

- `EMREVISAO`: o agente exibe sua G.U.I. (ver Figura 9.3) e aguarda alguma solicitação do revisor que poderá ser extrair o arquivo do artigo, redirecionar o AgenteFormRevisao, entrar com dados da revisão e finalizar a revisão.

Agente Formulário de Revisão

Dados sobre a Conferência

SBES - Simpósio Brasileiro de Engenharia de Software

R. Qualquer Coisa

De 16 á 16 de Maio de 2002

www.sbes.com.br

Artigo

Definição e Validação de Refatoramento de Software em Larga Escala

Extrair

Redirecionar

Endereço da Região: socket://10.6.40.32:7020/RegiaoConferencia

Nome da agência: agenciaFabiana

Retornar

Redirecionar

Dados do Revisor

Nome:

Instituição:

Endereço:

Telefone:

Fax:

Email:

Dados da Revisao

Originalidade: 5

Legibilidade: 5

Relevância: 5

Aceitação: 5

Comentários:

Comentários Sigilosos:

Salvar

Finalizar

Aprovar

Figura 9.3: G.U.I. do AgenteFormRevisao em revisão.

- EMAPROVACAO: o agente exibe sua G.U.I. (ver Figura 9.3) e aguarda alguma solicitação do revisor, que poderá ser extrair o arquivo do artigo, alterar dados da revisão e aprovar a revisão.

Figura 9.4: G.U.I. do AgenteFormRevisao em aprovação.

- `RETORNOU`: o *AgenteFormRevisao* cria um proxy do *AgenteCoordenador*, delega a ele a responsabilidade de registrar o resultado da revisão e se remove.

O método `migrar()` faz o *AgenteFormRevisao* migrar para o próximo destino segundo o seu itinerário. Quando não houver mais destinos no itinerário do agente, ele muda o seu `estado` para `RETORNOU` e volta para a agência do coordenador.

Código da classe *AgenteFormRevisao*:

```
Package agents;

Import de.ikv.grasshopper.agent.SmartAgent;
Import de.ikv.grasshopper.communication.GrasshopperAddress;
Import de.ikv.grasshopper.type.AgentSystemInfo;
```

```
Import de.ikv.grasshopper.util.SearchFilter;
Import de.ikv.grasshopper.type.Identifier;
Import de.ikv.grasshopper.communication.ProxyGenerator;

Import excecoes.MigrarException;
Import java.lang.reflect.Method;
Import java.lang.NoSuchMethodException;
Import java.lang.SecurityException;
Import java.lang.IllegalAccessException;
Import java.lang.reflect.InvocationTargetException;
Import java.sql.Date;

/**
 * Classe do Agente Formulário de Revisão que é do tipo móvel persistente. Este
 * agente é responsável por transportar os dados da revisão. Ele irá migrar
 * entre as máquinas do coordenador de programa, do membro de comitê e dos
 * revisores.
 * Este agente implementa uma variação do padrão Mestre-Escravo Itinerante.
 * @author  Fabiana Paulino Guedes
 * @version 1.0
 */
public class AgenteFormRevisao extends SmartAgent implements AgenteFormRevisaoIF
{

  /*
   * Todos os atributos do tipo transiente não fazem parte do estado de dados
   * do agente.
   */

  /** O estado assumido pelo agente logo após a sua criação. */
  transient final int CRIADOAGORA = 0;
  /**
   * O estado assumido pelo agente antes de migrar da agência do coordenador
   * de programa para a agência do membro de comitê onde ele deverá se clonar.
   */
  transient final int EMCLONAGEM = 1;
  /**
   * O estado assumido pelo agente antes de ser clonado. Indica que está
   * aguardo ser redirecionado  pelo membro de comitê para um revisor.
   */
  transient final int EMDISTRIBUICAO = 2;
  /**
   * O estado assumido pelo agente antes ser redirecionado pelo membro de
   * comitê para um revisor. Indica que o processo de revisão já foi iniciado,
   * mas que ainda está em aberto.
   */
  transient final int EMREVISAO = 3;
  /**
   * O estado assumido pelo agente após ter sido revisado.
   */
  transient final int EMAPROVACAO = 4;
  /**
   * O estado assumido pelo agente após ter retornado para agência do
   * coordenador de programa.
   */
  transient final int RETORNOU = 5;
  /**
   * GUI do Agente FormRevisao.
   */
  transient GuiAgenteFormRevisao gui;

  //Estado de dados do agente.
  /** Objeto com dados sobre a conferência */
  private DadosConferenciaIF dadosConferencia = null;
  /**
   * Objeto com dados sobre o registro de revisão do artigo. Este objeto é
   * composto pelos dados do Membro de Comitê responsável e pelo dados do artigo.
   */
```

```
private DadosRegistroRevisaoIF dadosRegistroRevisao = null;
/** Dados da Revisao */
private DadosRevisaoIF dadosRevisao = null;
/** Objeto com o itinérario do agente. */
private ItinerarioIF itinerario = null;
/** Endereço da Agência do Coordenador de Programa. */
private GrasshopperAddress endAgenciaCoordenador;
/** Estado atual do agente. */
int estado;
/** Estado anterior do agente. */
int estadoAnterior;
/**
 * Usado para sincronizar a noticação de espera.
 * Obs.: O objeto precisa ser serializável
 */
protected Object waitLock = new String();


/**
 * Obtém o nome do agente
 * @return Nome do Agente
 */
public String getName()
{
  return "Agente Formulário de Revisão";
}


/**
 * Inicializa o agente.
 * Argumentos de criação: arg[0] = dados da conferência
 *                        arg[1] = dados do registro de revisão
 * @param <B>args</B> Array com argumentos de criação.
 */
public void init(Object[] creationArgs)
{
  if (creationArgs.length < 2)
  {
    log("Argumentos de criação obrigatórios: <DadosConferenciaIF> "  +
        "<DadosRegistroRevisaoIF>");
    throw new RuntimeException();
  }
  try
  {
    /*
     * Obs.: não foi possível realizar o cast dos argumentos de criação,
     * assim fui obrigada a reconstruir os objetos através utilizando as
     * propriedades de reflexão Java.
     */
    // Reconstrói o objeto dadosConferencia a patir do primeiro argumento
    // de criação.
    dadosConferencia = criarDadosConferencia(creationArgs[0]);
    // Reconstrói o objeto dadosRegistroRevisao a patir do segundo argumento
    // de criação.
    dadosRegistroRevisao = criarDadosRegistroRevisao(creationArgs[1]);

    // Inicializa a tarefa do agente.
    inicializarTarefa();
  } catch (Exception e)
  {
    log("Erro: " + e.getMessage());
  }
}

/**
 * Contém o comportamento do agente. O AgenteFormRevisao executa sua tarefa,
 * que depende do valor do "estado" do agente, e depois migra para o póximo
 * destino, de acordo com o seu itinerário.
```

```
 */
public void live()
{
  try
  {
    log("Executando a  tarefa...");
    executarTarefa();
    log("Migrando para o próximo destino...");
    migrar();
  } catch(Exception e)
  {
    estado = estadoAnterior;
    log("Erro: " + e.getMessage());
  }
}

/**
 * Este método é executado antes da remoção do agente.
 */
public void beforeMove()
{
  destroiGui();
}

/**
 * O método action é invocado por um usuário ou por uma outra entidade de
 * software através da agência hospedeira.
 * Este método esta sendo sobrescrito e quando invocado exibirá a GUI do
 * agente.
 */
public void action()
{
  AgenteFormRevisaoIF proxyAgente =
  (AgenteFormRevisaoIF) getAgentProxy(this.getInfo(),
  AgenteFormRevisaoIF.class);
  Gui = new GuiAgenteFormRevisao(proxyAgente);
  Gui.show();
}

/**
 * Este método é automaticamente invocado antes do agente ser salvo
 * pela agência.
 */
public void beforeSave()
{
  log("Salvando minha memória...");
}

/**
 * Este método é automaticamente invocado antes do agente ser desativado
 * pela agência.
 */
public void beforeFlush()
{
  log("Estou sendo desativado...");
}

/**
 * Este método é automaticamente invocado depois do agente ser recarregado
 * pela agência.
 */
public void afterLoad()
{
  log("Ativando...");
}


/**
```

```
 * Inicializa a tarefa do agente.
 */
private void inicializarTarefa()
{
  log("Inicializar tarefa...");

  //Inicializa o estado do agente
  estado = CRIADOAGORA;

  //O endereço da agência do coordenador é igual ao home do agente,
  //mas precisamos guardá-la para que os clones deste agentes saibam para
  //onde retornar ao fim do processo de revisão, pois o home destes será
  //a agência do membro de comitê.
  EndAgenciaCoordenador = getInfo().getHome();

  //Obtém os dados do membro de comitê.
  DadosMembroComiteIF dadosMembroComite =
                                dadosRegistroRevisao.getDadosMembroComite();
  //Obtém o endereço da agência do Membro de Comite
  DestinoIF primeiroDestino = dadosMembroComite.getDestino();

  //Cria o itinerario do agente, já com o primeiro destino
  itinerario = new Itinerario(primeiroDestino);

  log("Fim de inicializar tarefa.");
}

/**
 * Este método controla as tarefas a serem realizadas pelo agente.
 * O agente executa uma determinada tarefa dependendo do valor do atributo
 * estado.
 */
private void executarTarefa() throws Exception
{

  log("Executar tarefa...");

  estadoAnterior = estado;

  switch (estado)
  {
    //O agente é recém criado
    case CRIADOAGORA:
    {
      //O estado muda para em clonagem antes da migração do agente para
      //a agência do membro de comitê, a fim de indicar que o agente
      //deve se clonar ao chegar em seu destino.
      estado = EMCLONAGEM;
      log("Criado com sucesso!!!");
      log("Migrando para a agência do membro de comitê...");
      break;
    }
    //O agente está na agência do membro de comitê e deve gerar as cópias
    //solicitadas.
    case EMCLONAGEM:
    {
      //Antes de se clonar o agente muda o seu estado para em distribuição,
      //a fim de evitar que os clones também gerem clones de si próprios e
      //assim sucessivamente.
      estado = EMDISTRIBUICAO;
      log("Gerando cópias solicitadas...");
      int clonesCriados = 0;
      try
      {
        //Gera clones do agente
        for (int i = 1; i < dadosRegistroRevisao.getQntCopias(); i++)
        {
          this.copy(this.getLocalizacao());
```

```
            clonesCriados++;
            log("Criei o clone: " + i);
          }
        } catch(Exception e)
        {
          //Caso ocorra alguma exceção durante a criação dos clones o
          // agente volta ao estado anterior antes da exceção.
          dadosRegistroRevisao.setQntCopias(
          (dadosRegistroRevisao.getQntCopias() - clonesCriados));
          estado = estadoAnterior;
          log("Erro ao clonar: " + e.getMessage());
        }
      }

      // O agente está na agência do membro de comitê esperando ser
      // redirecionado para um revisor.
      case EMDISTRIBUICAO:
      {
        log("Aguardando ser redirecionado para um revisor.");
        // Cria um proxy do agente
        AgenteFormRevisaoIF proxyAgente =
        (AgenteFormRevisaoIF) getAgentProxy(this.getInfo(),
        AgenteFormRevisaoIF.class);
        // Cria GUI de distribuição
        gui = new GuiAgenteFormRevisao(proxyAgente);
        gui.show();
        // Espera por um evento...
        waitForEvent();
        break;
      }

      case EMREVISAO:
      {
        log("#AgenteFormRevisao# Estou em processo de revisão...");
        // criar gui de revisao
        AgenteFormRevisaoIF proxyAgente =
        (AgenteFormRevisaoIF) getAgentProxy(this.getInfo(),
                                    AgenteFormRevisaoIF.class);
        gui = new GuiAgenteFormRevisao(proxyAgente);
        gui.show();
        // Espera por um evento...
        waitForEvent();

        break;
      }

      case EMAPROVACAO:
      {
        log("#AgenteFormRevisao# Estou em processo de aprovação...");

        // criar gui de revisao
        AgenteFormRevisaoIF proxyAgente =
        (AgenteFormRevisaoIF) getAgentProxy(this.getInfo(),
                                    AgenteFormRevisaoIF.class);
        gui = new GuiAgenteFormRevisao(proxyAgente);
        gui.show();
        // Espera por um evento...
        waitForEvent();
        break;
      }

      case RETORNOU:
      {
        log("Registrando resultados da revisão...");

        // Cria proxy do AgenteCoordenador
        SearchFilter filtro = new SearchFilter("NAME = AgenteCoordenador");
        AgenteMestreIF proxyAgenteCoordenador =
```

```
      (AgenteMestreIF) getAgentProxy(filtro,
      AgenteMestreIF.class);
      // Registra o resutado da revisão
      proxyAgenteCoordenador.registrarResultado(dadosRegistroRevisao,
                                          dadosRevisao);
      log("Serei removido...");
      removeMe();
    }
  }
}

/**
 * Redireciona o agente para uma outra agência.
 * @param <B>endRegiao</B> Endereço da região na qual a agência destino está
 * registrada. Formato: protocolo://nomeMaquina:numeroPorta/nomeRegiao. Ex.:
 * socket://10.6.40.32:7020/RegiaoConferencia.
 * @param <B>nomeAgencia</B> Nome da agência destino
 */
public void redirecionar(String endRegiao, String nomeAgencia, boolean retornar)
  throws MigrarException
{
  if (!retornar)
    itinerario.removerDestinoAtual();
  DestinoIF destino = new Destino(endRegiao, nomeAgencia);
  Itinerario.inserirDestino(destino);
  If (estado == EMDISTRIBUICAO)
    estado = EMREVISAO;
  notifyEvent();
}

/**
 * Atribui os dados do revisor e os dados da revisão.
 * @param <B>nomeRevisor</B> Nome do revisor
 * @param <B>instituicaoRevisor</B> Instituição do revisor
 * @param <B>endRevisor</B> Endereço do revisor
 * @param <B>foneRevisor</B> Telefone de contato do revisor
 * @param <B>faxRevisor</B> Fax do revisor
 * @param <B>emailRevisor</B> Endereço eletrônico do revisor
 * @param <B>originalidade</B> Nível de originalidade do artigo.
 * @param <B>legibilidade</B> Nível de legibilidade do artigo.
 * @param <B>relevancia</B> Nível de relevância do artigo.
 * @param <B>aceitacao</B> Nível de aceitação  do artigo.
 * @param <B>comentarios</B> Comentários adicionais.
 * @param <B>comentariosSigilosos</B> Comentários sigilosos.
 */
public void entrarDadosRevisao(String nomeRevisor, String instituicaoRevisor,
                               String endRevisor, String foneRevisor,
                               String faxRevisor, String emailRevisor,
                               String originalidade, String legibilidade,
                               String relevancia, String aceitacao,
                               String comentarios, String comentariosSigilosos)
{
  log("Entrando com os dados da revisão...");
  DestinoIF destinoAtual = itinerario.getDestinoAtual();
  If (dadosRevisao == null)
  {
    DadosRevisorIF dadosRevisor = criarDadosRevisor(nomeRevisor,
    instituicaoRevisor, endRevisor,
    foneRevisor, faxRevisor, emailRevisor,
    destinoAtual);
    dadosRevisao = criarDadosRevisao(originalidade, legibilidade, relevancia,
                  aceitacao, comentarios, comentariosSigilosos, dadosRevisor);
  } else
  {
    dadosRevisao.setOriginalidade(originalidade);
    dadosRevisao.setLegibilidade(legibilidade);
    dadosRevisao.setRelevancia(relevancia);
    dadosRevisao.setAceitacao(aceitacao);
```

```
      dadosRevisao.setComentarios(comentarios);
      dadosRevisao.setComentariosSigilosos(comentariosSigilosos);
      if (dadosRevisao.getDadosRevisor() == null)
      {
        DadosRevisorIF dadosRevisor = criarDadosRevisor(nomeRevisor,
                                      instituicaoRevisor, endRevisor, foneRevisor,
                                      faxRevisor, emailRevisor, destinoAtual);
        dadosRevisao.setDadosRevisor(dadosRevisor);

      } else
      {
        dadosRevisao.getDadosRevisor().setNome(nomeRevisor);
        dadosRevisao.getDadosRevisor().setInstituicao(instituicaoRevisor);
        dadosRevisao.getDadosRevisor().setEndereco(endRevisor);
        dadosRevisao.getDadosRevisor().setFone(foneRevisor);
        dadosRevisao.getDadosRevisor().setFax(faxRevisor);
        dadosRevisao.getDadosRevisor().setEmail(emailRevisor);
        dadosRevisao.getDadosRevisor().setDestino(destinoAtual);
      }
    }
    log("Fim da entrada dos dados da revisão");
  }

  /**
   * Finaliza o processo de revisão mudando o estado do agente para em aprovação.
   */
  public void finalizarRevisao()
  {
    log("Revisão finalizada.");
    estado = EMAPROVACAO;
    notifyEvent();
  }

  /**
   * Aprova a revisão.
   */
  public void aprovarRevisao()
  {
    log("Revisão aprovada.");
    // Notifica ao agente que um evento ocorreu para que o seu método live()
    // possacontinuar
    notifyEvent();
  }

  /**
   * Extrai o arquivo do artigo a ser revisado em um determinado caminho.
   * @param <B>caminho</B> diretorio/nomeArquivo.pdf
   */
  public void extrairArquivo(String caminho)
  {
    try
    {
      log("Salvando como:" + caminho);
      dadosRegistroRevisao.getDadosArtigo().extrair(caminho);
      log("Fim da extração do arquivo.");
    } catch (Exception e)
    {
      log("Exception: ", e);
    }

  }

  /**
   * Migra para o próximo destino conforme o seu itinerario.
   * Se o itinerario estiver vazio o agente retorna para a agência do
   * coordenador de programa.
   */
  private void migrar() throws MigrarException
```

```
{
  try
  {
    DestinoIF proximoDestino = itinerario.getProximoDestino();
    if (proximoDestino != null)
    {
      log("Migrando para o próximo destino...");
      log("Região: " + proximoDestino.getEndRegiao());
      log("Nome Agencia: " + proximoDestino.getNomeAgencia());
      migrar(proximoDestino);
    }
    else
    {
      log("Retornando para a agência do coordenador de programa...");
      estado = RETORNOU;
      move(endAgenciaCoordenador);
    }
  } catch (Exception e)
  {
    throw new MigrarException("Falhou ao migrar para o próximo destino :" +
                              e.getMessage());
  }
}

/**
 * Migra para um destino determinado.
 * @param <B>destino</B> Destino
 */
public void migrar(DestinoIF destino) throws Exception
{

  //Recupera o endereço da agência destino
  GrasshopperAddress endRegiao =
    new GrasshopperAddress(destino.getEndRegiao());
  SearchFilter filtro =
    new SearchFilter("NAME = " + destino.getNomeAgencia());
  AgentSystemInfo[] agencia =
    (this.getRegion()).listAgencies(endRegiao, filtro);
  GrasshopperAddress endAgencia = agencia[0].getLocation();

  Log("Migrar para agencia: " + endAgencia +
      " da região: " + endRegiao);
  move(endAgencia);
}

/**
 * Obtém os dados da Conferência.
 * @return Dados da Conferência.
 */
public DadosConferenciaIF getDadosConferencia()
{
  return dadosConferencia;
}

/**
 * Obtém os dados do registro de revisão do artigo.
 * @return Dados do registro de revisão do artigo.
 */
public DadosRegistroRevisaoIF getDadosRegistroRevisao()
{
  return dadosRegistroRevisao;
}

/**
 * Obtém os dados da revisão do artigo.
 * @return Dados da revisão do artigo.
 */
public DadosRevisaoIF getDadosRevisao()
```

```
{
  return dadosRevisao;
}

/**
 * Obtém estado do agente.
 * @return Estado do agente.
 */
public int getEstado()
{
  return estado;
}

/**
 * Obtém a localização do agente.
 * @return Localização do agente.
 */
public GrasshopperAddress getLocalizacao()
{
  return (this.getInfo()).getLocation();
}

/**
 * Destrói a GUI do agente.
 */
private void destroiGui()
{
  if (gui!=null)
  {

    log("Destrói a gui");
    //Oculta e detrói a gui do agente
    gui.hide();
    gui.dispose();
    gui = null;
  }
}

/**
 * Factory method. Cria o objeto DadosRevisão.
 */
private DadosRevisaoIF criarDadosRevisao(String originalidade,
        String legibilidade, String relevancia, String aceitacao,
        String comentarios, String comentariosSigilosos,
        DadosRevisorIF dadosRevisor)
{
  return new DadosRevisao(originalidade, legibilidade, relevancia, aceitacao,
                          comentarios, comentariosSigilosos, dadosRevisor);

}
/**
 * Factory method. Cria o objeto DadosRevisor.
 */
private DadosRevisorIF criarDadosRevisor(String nomeRevisor,
        String instituicaoRevisor, String endRevisor,
        String foneRevisor, String faxRevisor,
        String emailRevisor, DestinoIF destino)
{
  return new DadosRevisor(nomeRevisor, instituicaoRevisor, endRevisor,
                          foneRevisor, faxRevisor, emailRevisor, destino);
}

/**
 * Factory method. Cria o objeto DadosConferencia.
 */
private DadosConferenciaIF criarDadosConferencia(Object objeto)
        throws NoSuchMethodException, IllegalAccessException,
               InvocationTargetException
```

```
{
  Class  classe  = objeto.getClass();
  Method metodo = classe.getDeclaredMethod("getTitulo", new Class[0]);
  String titulo = (String)metodo.invoke(objeto, new Object[0]);
  Metodo = classe.getDeclaredMethod("getDtInicio", new Class[0]);
  Date dtInicio = (Date)metodo.invoke(objeto, new Object[0]);
  Metodo = classe.getDeclaredMethod("getDtFinal", new Class[0]);
  Date dtFinal = (Date)metodo.invoke(objeto, new Object[0]);
  Metodo = classe.getDeclaredMethod("getEndereco", new Class[0]);
  String endereco = (String)metodo.invoke(objeto, new Object[0]);
  Metodo = classe.getDeclaredMethod("getUrl", new Class[0]);
  String url = (String)metodo.invoke(objeto, new Object[0]);
  Return new DadosConferencia(titulo, dtInicio, dtFinal, endereco, url);
}

/**
 * Cria um objeto do tipo DadosRegistroRevisaoIF através de reflexão.
 */
private DadosRegistroRevisaoIF criarDadosRegistroRevisao(Object objeto)
        throws NoSuchMethodException, IllegalAccessException,
               InvocationTargetException
{
  Class  classe  = objeto.getClass();
  Try
  {
    Method[] metodos = classe.getDeclaredMethods();
    int i;
    for(i=0; i < metodos.length; i++)
    {
      log(metodos[i].toString());
    }
  } catch (SecurityException e)
  {
  }
  Method metodo = classe.getDeclaredMethod("getId", new Class[0]);
  Int id = ((Integer)metodo.invoke(objeto, new Object[0])).intValue();
  Metodo = classe.getDeclaredMethod("getDadosArtigo", new Class[0]);
  Object objArtigo = metodo.invoke(objeto, new Object[0]);
  DadosArtigoIF dadosArtigo = criarDadosArtigo(objArtigo);
  Metodo = classe.getDeclaredMethod("getDadosMembroComite", new Class[0]);
  Object objMembroComite =  metodo.invoke(objeto, new Object[0]);
  DadosMembroComiteIF dadosMembroComite =
    criarDadosMembroComite(objMembroComite);
  metodo = classe.getDeclaredMethod("getDtEnvio", new Class[0]);
  Date dtEnvio = (Date)metodo.invoke(objeto, new Object[0]);
  Metodo = classe.getDeclaredMethod("getQntCopias", new Class[0]);
  Int qntCopias = ((Integer)metodo.invoke(objeto, new Object[0])).intValue();
  Return new DadosRegistroRevisao(id, dadosArtigo, dadosMembroComite,
                                  dtEnvio, qntCopias);
}

/**
 * Cria um objeto do tipo DadosArtigoIF através de reflexão.
 */
private DadosArtigoIF criarDadosArtigo(Object objeto)
throws NoSuchMethodException, IllegalAccessException,
InvocationTargetException
{

  Class  classe  = objeto.getClass();
  Method metodo = classe.getDeclaredMethod("getId", new Class[0]);
  Int id = ((Integer)metodo.invoke(objeto, new Object[0])).intValue();
  Metodo = classe.getDeclaredMethod("getTitulo", new Class[0]);
  String titulo = (String)metodo.invoke(objeto, new Object[0]);
  Metodo = classe.getDeclaredMethod("getArquivo", new Class[0]);
  Byte[] arquivo = (byte[])metodo.invoke(objeto, new Object[0]);
  Return new DadosArtigo(id, titulo, arquivo);
}
```

```
/**
 * Cria um objeto do tipo DadosMembroComiteIF através de reflexão.
 */
private DadosMembroComiteIF criarDadosMembroComite(Object objeto)
throws NoSuchMethodException, IllegalAccessException,
InvocationTargetException
{

  Class  classe  = objeto.getClass();
  Method metodo = classe.getDeclaredMethod("getId", new Class[0]);
  Int id = ((Integer)metodo.invoke(objeto, new Object[0])).intValue();
  Metodo = classe.getDeclaredMethod("getNome", new Class[0]);
  String nome = (String)metodo.invoke(objeto, new Object[0]);
  Metodo = classe.getDeclaredMethod("getEndRegiao", new Class[0]);
  String endRegiao = (String)metodo.invoke(objeto, new Object[0]);
  Metodo = classe.getDeclaredMethod("getNomeAgencia", new Class[0]);
  String nomeAgencia = (String)metodo.invoke(objeto, new Object[0]);

  Return new DadosMembroComite(id, nome, endRegiao, nomeAgencia);
}

/**
 * Espera até que alguma coisa especial aconteça.
 * Use este método para bloquear o método life() até que alguma coisa
 * aconteça.
 */
protected void waitForEvent()
{
  waitForEvent(-1);
}

/**
 * Espera durante um certo tempo até que algum evento seja notificado.
 * Use este método para bloquear o método life() durante um certo tempo
 * até que algum evento seja notificado. Se timeout igual a -1 espera pela
 * notificação de um evento indefinidamente.
 * @param <B>timeout</B> Tempo em  milesegundos.
 */
protected void waitForEvent(long timeout)
{

  synchronized(waitLock)
  {
    try
    {
      if (timeout == -1)
      {
        waitLock.wait(); // wait until we are notified
      }
      else
      {
        waitLock.wait(timeout);
      }
    }
    catch(InterruptedException e)
    {
      log("interrupted");
    }
  }
}

/**
 * Chame este método de qualquer outra thread para notificar o agente.
 * Por exemplo, no método action();
 */
protected void notifyEvent()
{
```

```
    synchronized(waitLock)
    {
      waitLock.notify();
    }
  }
}
```

9.9 Interface ItinerarioIF

Interface que define os métodos do itinerário de um agente.

Código da interface *ItinerarioIF*:

```
package agentes;

import excecoes.MigrarException;

/**
 * Interface do Itinerário de um agente.
 * @author  Fabiana Paulino Guedes
 * @version 1.0
 */
public interface ItinerarioIF
{

  /**
   * Obtém o destino atual.
   * @return Destino atual do itinerário.
   */
  public DestinoIF getDestinoAtual();

  /**
   * Obtém o próximo destino.
   * @return Próximo destino do itinerário.
   */
  public DestinoIF getProximoDestino();

  /**
   * Obtém o tamanho do itinerário.
   * @return Tamanho do itinerário.
   */
  public int tamanho();

  /**
   * Insere um novo destino no itinerário.
   * @param Novo destino.
   */
  public void inserirDestino(DestinoIF destino);

  /**
   * Remove o destino atual do itinerário.
   */
  public void removerDestinoAtual();

}
```

9.10 Classe Itinerario

Classe que representa o itinerário do agente e tem como responsabilidade determinar o próximo destino do agente.

No padrão Itinerário proposto por Lange et al em [LO98], a responsabilidade de migração do agente é do objeto itinerário que deveria oferecer em sua interface o método `migrar()`. Este método deveria determinar o próximo destino do agente e fazê-lo migrar para lá. Para tal, o itinerário precisaria invocar o método de migração do agente (no caso da plataforma Grasshopper o método `move(...))` através de um proxy do mesmo. Inicialmente, implementamos este método em nossa classe Itinerario, porém depois que ele era invocado o sistema travava. Procurando uma razão para tal comportamento, descobrimos na documentação da Plataforma Grasshopper que não era aconselhável chamar os métodos `move()` e `copy()` do agente fora da sua thread principal, ou seja, do método `live()`, a fim de evitar travamentos. Assim, resolvemos alterar a definição da nossa classe Itinerario tirando dela a responsabilidade de migração do agente e permanecendo apenas a responsabilidade de determinar o próximo destino.

O próximo destino do agente pode ser obtido através do método `getProximoDestino()`. Para determinar qual será o próximo destino do agente, o itinerario verifica se existe um destino seguinte ao atual no itinerario, caso exista, este será o próximo destino e caso contrário, será o destino anterior, o que significa que o agente chegou ao fim do seu itinerário e tem que retornar. Caso não exista destino anterior ao atual, o método `getProximoDestino()` retorna `null`.

Código da classe *Itinerario*:

```
Package agentes;

import excecoes.MigrarException;

import java.net.*;
import java.io.*;
import java.util.Vector;

/**
 * Esta classe representa o itinerario do agente. Ela é reponsável por determinar
 * o próximo destino do agente.
 * @author  Fabiana Paulino Guedes
 * @version 1.0
 */
public class Itinerario implements ItinerarioIF, Serializable
{

  /** Vetor com os destinos do agente */
  private Vector destinos;
  /** Índice do destino atual */
  private int indiceDestinoAtual = -1;
```

```
  /**
   * Cria um novo objeto Itinerario.
   * @param <B>primeiroDestino</B> Primeiro destino do itinerário.
   */
  public Itinerario(DestinoIF primeiroDestino)
  {
    this.destinos = new Vector();
    inserirDestino(primeiroDestino);
  }

  /**
   * Obtém o destino atual.
   * @return Destino atual do itinerário.
   */
  public DestinoIF getDestinoAtual()
  {
    if (indiceDestinoAtual < 0)
      return null;
    return (DestinoIF)destinos.get(indiceDestinoAtual);
  }

  /**
   * Obtém o próximo destino.
   * @return Próximo destino do itinerário.
   */
  public DestinoIF getProximoDestino()
  {
    DestinoIF destino = getDestinoSeguinte();
    if (destino != null)
    {
      indiceDestinoAtual = indiceDestinoAtual + 1;
    } else
    {
      destino = getDestinoAnterior();
      if ( destino != null)
      {
        System.out.println("Retornando");
        removerDestinoAtual();
      }
    }
    return destino;
  }

  /**
   * Obtém o tamanho do itinerário.
   * @return Tamanho do itinerário.
   */
  public int tamanho()
  {
    return destinos.size();
  }

  /**
   * Insere um novo destino no itinerário.
   * @param Novo destino.
   */
  public void inserirDestino(DestinoIF destino)
  {
    destinos.add(indiceDestinoAtual + 1, destino);
  }

  /**
   * Remove o destino atual do itinerário.
   */
  public void removerDestinoAtual()
  {
    removerDestino(indiceDestinoAtual);
    indiceDestinoAtual = indiceDestinoAtual - 1;
```

```
  }

  /**
   * Obtém destino seguinte ao atual no vetor de destinos.
   * @return Destino seguinte ao atual no vetor de destinos.
   */
  private DestinoIF getDestinoSeguinte()
  {
    if (indiceDestinoAtual + 1 < this.tamanho())
    {
      return (DestinoIF)destinos.get(indiceDestinoAtual + 1);
    } else
    {
      return null;
    }
  }

  /**
   * Obtém destino anterior ao atual no vetor de destinos.
   * @return Destino anterior ao atual no vetor de destinos.
   */
  private DestinoIF getDestinoAnterior()
  {
    System.out.println("IndiceAtual: " + indiceDestinoAtual);
    if (indiceDestinoAtual > 0)
    {
      return (DestinoIF)destinos.get(indiceDestinoAtual - 1);
    } else
    {
      return null;
    }
  }

  /**
   * Remove o destino de índice <B>i</B> do itinerário
   * @param <B>i</B> Índice
   */
  private DestinoIF removerDestino(int i)
  {
    return (DestinoIF)destinos.remove(i);
  }
}
```

10 REFERÊNCIAS BIBLIOGRÁFICAS

[Bra00] Braga, André Luiz. Utilização de agentes móveis em recuperação e troca de dados. Proposta de Tese submetida ao corpo docente da Coordenação dos Programas de Pós-Graduação de Engenharia da Universidade Federal do Rio de Janeiro, como Exame de Qualificação para a obtenção do Grau de Doutor em Ciências em Engenharia de Sistemas e Computação, Rio de Janeiro-RJ, Brasil, Setembro de 2000.

[BHM98] Breugst, M.; Hagen, L; Magedanz, T.. Impacts of Mobile Agent Technology on Mobile Communications System Evolution. IEEE Personal Communication Magazine, Agosto de 1998.

[BHT98] Breugst, M.; Hagen, L; Magedanz, T.. Impacts of Mobile Agent Technology on Mobile Communications System Evolution. IEEE Personal Comunication Magazine, Agosto de 1998.

[BM00] Bernardes, M. C.; Moreira, E. S. Implementation of an intrusion detection system based on mobile agents. *In* International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000). IEEE, 2000.

[BP98] Blaha, M. R.; Premerlani, W. Object-Oriented Modeling and Design for Database Applications. Prentice Hall, 1998.

[Car99] L. Cardelli. Abstractions for mobile computation. *In* Proceedings of Secure Internet Programming, Vol. 1603 of Lecture Notes in Computer Science, 1999.

[CLZ01] Cabri, G.; Leonardi, L; Zambonelli, F. Engineering mobile-agent applications via context-dependent coordination. *In* Proceedings of International Conference on Software Engineering - ICSE'2001, 2001.

[FPV98] Fuggeta, A.; Picco, G.P.; Vigna, G. Understanding Code Mobility, IEEE Transactions on Software Engineering. Vol. 24, No. 5, Maio de 1998.

[GHJV95] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns. Reading, MA.: Addison-Wesley, 1995

[Glas] Glass, G. ObjectSpace Voyager Core Package Technical Overview. ObjectSpace, White Paper. Também em [Milojičić, 1999], pp. 612-627.

[GSLM01] Garcia, Alessandro F., Silva, Viviane T., Lucena, Carlos J.P., Milidiú, Ruy L. An Aspect Based Object Oriented Model for Multi-Agent Systems. *In* XVI SBES – Simpósio de Engenharia de Software. Rio de Janeiro, 2001.

[GM01] Guedes, F. P.; Machado, P. D. L.. Um modelo para o desenvolvimento de aplicações baseadas em agentes móveis. *In* VI WTES – Workshop de Teses em Engenharia de Software. Rio de Janeiro, 2001.

[GOP02] Glitho, R. H.; Olougouna, E.; Pierre S. Mobile agents and their use for information retrieval: A brief overview and an elaborate case study. IEEE Network, 2002.

[Gro97] The Object Management Group. The Mobile Agent System Interoperability Facility. The Object Management Group, Framingham, MA, 1997.

[HB99] Hayzelden and Bigham, editors. Agents for Future Communication Systems. Springer,1999.

[HCMKK00] Horvat, D.; Cvektović, D.; Milutinović; Kočović, P.; Kovačević, V. Mobile Agents and Mobile Agents Tookits. Procceedings of 33d Hawaii International Conference on System Sciences–2000, Maui, Hawai'i, USA, Janeiro de 2000.

[IKV01] IKV++ GmbH. Grasshopper Programmer's Guide, Release 2.2, Germany, 2001. http://www.grasshopper.de

[KRSW01] Klein C.; Rausch, A.; Sihling, M.; Wen, Z. Extension of the Unified Modeling Language for mobile agents. *In* Keng Siau and Terry Halpin, editors, Unified Modeling Language: Systems Analysis, Design and Development Issues, Cap. 8, p. 116–128. Idea Publishing Group, 2001.

[Lar02] Larman, Craig. Applying UML and patterns an introduction to object-oriented analysis and design. Prentice-Hall, Inc, 2002.

[LO98] Lange, D.; Oshima, M. Mobile agents with Java: The Aglet API.

World Wide Web, 1(3), Setembro1998. Também em [MDW99], p. 495-512.

[MDW99] Milojičić, D.; Douglis, F.; Wheeler, R. Mobility: process, computers and agents. Association for Computing Machinery, Inc. (ACM), 1999.

[MEI98] Mitsubish Eletric ITCA. Mobile Agent Computing. WhitePaper, January 19, 1998.

[MKC01] Mylopoulos, J.; Kolp, M.; Castro, J. UML for agent-oriented software development: The tropos proposal. *In* M. Gogolla and C. Kobryn, editors, Proceedings of UML 2001, Vol. 2185 of Lecture Notes in Computer Science, p. 422–441, 2001.

[OPB00] Odell, J.; Parunak, V. D.; Bauer, B.. Extending UML for agents. *In* Proceedings of the Agent-Oriented Information System Workshop at the 17th National Conference on Artificial Intelligence, p. 3–17, Austin, USA, Julho de 2000.

[Poo01] Poole, John D. Model-Driven Architecture: Vision, Standards And Emerging Technologies, *In* Workshop on Metamodeling and Adaptive Object Models, ECOOP2001, 2001.

[Sil99] Silva, A. Rodrigues. O Impacto dos Agentes no Futuro da Internet, Sistemas e Tecnologias de Informação: Desafios para o Século XXI. Universidade Católica Portuguesa, Lisboa. Outubro de 1999.

[TOH99] Tahara, Y.; Ohsuga, A.; Honiden, S. Agent system development method based on agent patterns. *In* Proceedings of International Conference on Software Engineering-ICSE'99, 1999.

[VBB00] Valente, M. T.; Bigonha, R.; Loureiro, A. A.; Bigonha, M. Linguagens para computação móvel na Internet (tutorial). Em *IV* Simpósio Brasileiro de Linguagens de Programação. *Sociedade* Brasileira de Computação, Maio 2000.

[VBML99] Valente, T. M.; Bigonha, R.; Maia, M.; Loureiro, A. Especificacão formal de agentes móveis usando máquinas de estado abstratas. *In* Workshop de Comunicação Sem Fio, UFMG, 1999.

[VZM00] Venieris, Zizza, and Magedanz, editors. Object Oriented Software Technologies in Telecomunications. John Wiley & Sons, 2000.

[WJK99] Wooldridge, M.; Jennings, N.; Kinny, D. A methodology for agent-oriented analysis and design. *In* Proceedings of the Third International Conference on Autonomous Agents-Agents'99, 1999.

[Woo99] Wooldridge, M. Intelligent agents. In G. Weiss, editor, *Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999.

[YBF98] Yoo, M. J.; Briot, J. P.; Ferber, J. Using components for modeling intelligent and collaborative mobile agents. *In* Proceedings of the Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, WET ICE 1998. IEEE, 1998.

[Yu95] Yu, E.. Modelling Strategic Relationships for Process Reengineering. Ph.D. thesis, Department of Computer Science, University of Toronto, Canada, 1995