

**UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA**



-:[kAgent]:-

**UMA API JAVA PARA AGENTES INTELIGENTES
EM DISPOSITIVOS MÓVEIS**

RYAN LEITE ALBUQUERQUE

-:[2002]:-

**UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA**

RYAN LEITE ALBUQUERQUE

kAgent

***Uma API Java para Agentes Inteligentes em
Dispositivos Móveis***

Este trabalho foi submetido à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação

ORIENTADOR: GEBER LISBOA RAMALHO

RECIFE, 30 DE AGOSTO DE 2002



ÍNDICE DE TÓPICOS

1.	INTRODUÇÃO	1
1.1.	ATUALIDADE E FUTURO PRÓXIMO	1
1.2.	UBIQUIDADE	2
1.3.	J2ME	3
1.4.	AGENTES	4
1.5.	OBJETIVOS	4
1.6.	ABORDAGEM E CONTRIBUIÇÃO	5
1.7.	PRÓXIMOS CAPÍTULOS	6
2.	AGENTES EM DISPOSITIVOS MÓVEIS	8
2.1.	AMIGABILIDADE E INTELIGÊNCIA	9
2.2.	FUNCIONALIDADES DESEJÁVEIS	11
2.3.	REQUISITOS COMPUTACIONAIS	13
2.3.1.	Componente de Raciocínio	14
2.3.2.	Componente de Comunicação	16
2.4.	CONCLUSÃO	17
3.	LINGUAGEM DE PROGRAMAÇÃO	18
3.1.	REQUISITOS	18
3.1.1.	Portabilidade	19
3.1.2.	Memória e Processamento	19
3.1.3.	Comprometimento da indústria	19
3.1.4.	Facilidades	20
3.1.5.	Suporte a Agentes	20
3.2.	ABORDAGENS PARA EMBUTIR AGENTES NOS DISPOSITIVOS	20
3.2.1.	Acesso Nativo	21
3.2.2.	Máquina Virtual	22
3.3.	CONCLUSÃO	24
4.	J2ME	25
4.1.	HISTÓRICO	25
4.2.	ARQUITETURA DE J2ME	27
4.2.1.	KVM (Máquina virtual)	27
4.2.2.	Configuração	28
4.2.3.	Perfil (Profile)	31
4.3.	PROCESSO DE DISTRIBUIÇÃO DAS APLICAÇÕES	34
4.4.	PROCESSO DE INSTALAÇÃO DAS APLICAÇÕES	35
4.5.	ADERÊNCIA MERCADOLÓGICA	37
4.6.	CONCLUSÕES	38
5.	COMUNICAÇÃO	39

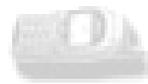
5.1.	LINGUAGENS DE COMUNICAÇÃO	39
5.2.	REQUISITOS	40
5.3.	KSE	41
5.3.1.	<i>Ontologia</i>	42
5.3.2.	<i>Linguagem Interna</i>	43
5.3.3.	<i>Linguagem Externa</i>	44
5.3.4.	<i>KQML</i>	44
5.4.	FIPA	51
5.5.	FERRAMENTAS	53
5.5.1.	<i>Saci</i>	54
6.	KSACI	56
6.1.	SACI	56
6.1.1.	<i>Modelo de Sociedade SACI</i>	57
6.1.2.	<i>Arquitetura SACI</i>	59
6.2.	PROCESSO DE EXTENSÃO	63
6.2.1.	<i>Dificuldades</i>	64
6.2.2.	<i>De RMI Para Http</i>	64
6.2.3.	<i>Serialização</i>	69
6.3.	FUNCIONAMENTO	72
6.3.1.	<i>Exemplo 1</i>	73
6.3.2.	<i>Exemplo 2</i>	76
7.	RACIOCÍNIO	79
7.1.	DEDUÇÃO EM DISPOSITIVOS MÓVEIS	79
7.1.1.	<i>LPOIA Embutida no Sistema Operacional Hospedeiro</i>	80
7.1.2.	<i>Interfaces para Interoperabilidade</i>	81
7.1.3.	<i>Linguagem Hospedeira para Construção de APIs</i>	82
7.2.	EOOPS EM DISPOSITIVOS MÓVEIS	83
7.2.1.	<i>Sistemas de Produção</i>	83
7.2.2.	<i>EOOPS</i>	85
7.2.3.	<i>EOOPS e J2ME</i>	93
7.3.	FERRAMENTAS A SEREM REUSADAS	95
7.3.1.	<i>JESS</i>	96
7.3.2.	<i>JEOPS</i>	98
7.4.	CONCLUSÃO	99
8.	KEOPS	101
8.1.	JEOPS	101
8.1.1.	<i>Sintaxe Regras</i>	102
8.1.2.	<i>Máquina de Inferência</i>	104
8.1.3.	<i>Processo de Desenvolvimento</i>	105
8.2.	TRABALHO DE ADAPTAÇÃO: KEOPS	106
8.2.1.	<i>Dificuldades</i>	107
8.2.2.	<i>Unificação Comportamental Sem Reflexão</i>	108
8.3.	FUNCIONAMENTO	113
8.3.1.	<i>Exemplos</i>	113
8.4.	RESULTADOS	117
9.	CONCLUSÃO	119
APÊNDICE A: ATOS DE FALA		123
9.1.	TAXONOMIA	125
10. REFERÊNCIAS		127



ÍNDICE DE FIGURAS

Figura 1.1: Crescimento do Número de Assinantes de Serviços 3G (Fonte: Telecompetitions Inc, Fevereiro de 2001).....	2
Figura 2.1: Crescimento do Rendimento dos Serviços 3G. (Fonte: Telecompetitions Inc, Fevereiro de 2001).....	8
Figura 2.2: Arquitetura de raciocínio dedutivo. Tell adiciona sentenças à BC. Ask consulta a BC.....	14
Figura 3.1: Uso de Máquina Virtual.	22
Figura 4.1: Crescimento no tamanho da linguagem Java	26
Figura 4.2: Modelo de Camadas de J2ME	27
Figura 4.3: Cobertura de classes de CDC e CLDC em relação à J2SE	31
Figura 4.4: Atual arquitetura de Java [45]	34
Figura 4.5: Processo de distribuição de uma aplicação	35
Figura 4.6: Processo de instalação de programas em dispositivos	36
Figura 4.7: Exemplos de dispositivos que suportam J2ME	37
Figura 5.1: Camadas da linguagem KQML	44
Figura 5.2: Componentes de uma mensagem KQML	47
Figura 5.3: Modelo de um roteador para interface entre agentes e a rede	48
Figura 5.4: Comunicação usando protocolo simples ponto-a-ponto	49
Figura 5.5: Uso da primitiva <i>subscribe</i>	49
Figura 5.6: Uso da primitiva <i>broker-one</i>	50
Figura 5.7: Uso da primitiva <i>recruit</i>	51
Figura 6.1: Arquitetura SACI	59
Figura 6.2: Troca de mensagens utilizando o facilitador	61
Figura 6.3: Anúncio de habilidades	63
Figura 6.4: Arquitetura KSACI	65
Figura 6.5: kAgent Diagram	66
Figura 6.6: kMailbox Diagram	67
Figura 6.7: kMessage Diagram	68
Figura 6.8: Rede de dependências entre objetos Java	70
Figura 6.9: Diagrama da classe MappedClass	70
Figura 6.10: Diagrama da classe Unmarshaller	72
Figura 6.11: Diagrama da classe Marshaler	72
Figura 6.12: Telas do MIDlet que soma dois números	75
Figura 6.13: Output da execução no Sun J2ME Wireless Toolkit	75
Figura 6.14: Ambiente de monitoração do SACI	76
Figura 7.1: Linguagem orientada a IA embutida no sistema operacional hospedeiro	80
Figura 7.2: Interfaces para interoperabilidade entre sistemas	81
Figura 7.3: Tradutor de linguagens	82
Figura 7.4: Embutindo mecanismo de raciocínio numa linguagem hospedeira	83
Figura 7.5: Compilação de um programa da linguagem C	90
Figura 7.6: Compilação de um programa da linguagem Java	90
Figura 7.7: Pré-compilação de regras	91

Figura 7.8: Interpretação de regras	92
Figura 7.9: Exemplo de Herança	94
Figura 8.1: Processo de desenvolvimento de uma aplicação JEOPS	106
Figura 8.2: O usuário informa explicitamente os <i>meta-dados</i> das classes.	109
Figura 8.3: O compilador JEOPS gera as informações necessárias.	110
Figura 8.4: Um pós-processador gera as informações necessárias.	110
Figura 8.5: Processo de desenvolvimento de uma aplicação KEOPS	112
Figura 8.6: Pós-Processador (Classe Serializer)	113



INTRODUÇÃO

1.1. ATUALIDADE E FUTURO PRÓXIMO

Este novo século está testemunhando um crescimento sem precedentes da indústria de dispositivos móveis que, de acordo com estudos recentes (Figura 1.1), conta com um número de usuários em nível mundial na ordem de 8 milhões em 2002 e terá algo em torno de 1,4 bilhão até o final de 2010. Este número, se comparado a base de PCs¹ instalados até a virada do século (350 milhões), é algo surpreendente [01]. Este crescimento será ainda mais acentuado com o surgimento da 3G *Wireless Broadband Network*, a rede sem fio de terceira geração baseada em pacotes que possui uma maior banda passante (384-2Mbps), comparados aos 19.2Kbps das redes 2G. Com o aumento da banda passante para os dispositivos móveis, os usuários não terão somente maior espaço para ligações telefônicas, mas também uma viabilização de serviços de áudio, vídeo e aplicações mais complexas, o que engatilhará uma explosão da Internet sem fio e uma demanda de aplicações sem igual.

¹ Personal Computer

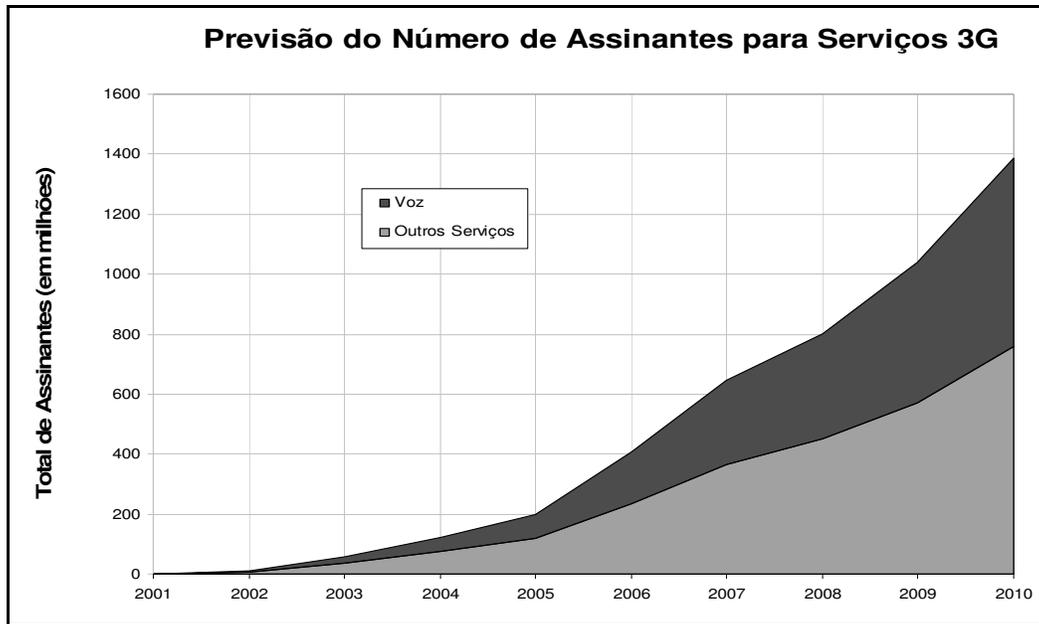


Figura 1.1: Crescimento do Número de Assinantes de Serviços 3G
 (Fonte: Telecompetitions Inc, Fevereiro de 2001)

1.2. UBIQUIDADE

Nesta revolução, os dispositivos móveis irão desempenhar um papel fundamental. De fato, estes dispositivos estão deixando de ser somente aparelhos para realizar ligações telefônicas para serem pequenos computadores ambulantes. Várias funcionalidades extras estão sendo incorporadas a estes dispositivos, fazendo com que eles se tornem cada vez mais importantes em nossas vidas. Hoje em dia, as pessoas já usam seus celulares e/ou PDAs (*Personal Digital Assitents*, mais conhecidos como palm-tops) para acessar informações em tempo real, tais como notícias, valores do mercado de ações e previsões climáticas, a maioria provida por tecnologias como WAP [02] e iMode [03]. Com o surgimento da banda larga e com o aperfeiçoamento dos processadores, a computação ubíqua [04] começará a tomar forma. As pessoas não estarão mais satisfeitas em *surf*ar na *Internet* através de seus celulares. Elas vão provavelmente querer ter serviços computacionais a toda hora e em todo lugar; aplicações mais interativas, inteligentes e que melhorem significativamente

seu modo de vida e simplifiquem a forma como elas trabalham, negociam, se locomovem, etc.

1.3. J2ME

Um dos esforços para implementação da desejada computação ubíqua é a recente especificação da Edição Micro da Plataforma Java, comumente conhecida como J2ME [05] ou KJAVA, uma linguagem de programação fruto de um consórcio da *Sun Microsystems* com os principais fabricantes de dispositivos de telecomunicações (e.g. *Motorola, Nokia, Sony, Samsung e Ericsson*). J2ME foi desenhada especialmente para o desenvolvimento de aplicações que rodam em dispositivos móveis, levando em conta todas as limitações de memória, processamento e espaço para armazenamento. Como as edições anteriores (J2SE e J2EE), a J2ME possui todas as características essenciais de orientação a objetos e portabilidade. As diferenças ficam em torno da adequação dos serviços de rede e interface com o usuário, assim como no tamanho reduzido da API. De fato, uma série de funcionalidades das versões anteriores não poderiam estar disponíveis aos desenvolvedores, pelo menos com as atuais configurações dos dispositivos móveis.

J2ME expande as perspectivas do mercado, uma vez que permite os desenvolvedores de aplicações produzirem seus próprios softwares para este tipo de dispositivo, o que era praticamente inviável há pouco tempo atrás, uma vez que as plataformas eram todas proprietárias. Além do próprio fabricante, ninguém poderia escrever uma aplicação para executar dentro de um dispositivo móvel. Outra grande vantagem da linguagem J2ME é que, como Java, ela é ideal para o desenvolvimento de aplicações independentes de plataforma, uma vez que se baseia no uso de uma máquina virtual que por sua vez proporciona uma interface única para todas as plataformas de dispositivos.

1.4. AGENTES

Com a prevista explosão da demanda de aplicações, acentuada ainda mais com o surgimento da J2ME e da rede sem fio de terceira geração (3G), não é difícil imaginar que existirão aplicações dos mais variados tipos: simples como sistemas para enviar mensagens curtas (SMS) e agendas despertadoras; e complexas como jogos interativos multi-jogadores, assistentes pessoais inteligentes e sistemas de transações comerciais sensíveis à localização do usuário [06].

Obviamente, nem todos estes problemas podem ou devem ser completamente resolvidos com as tecnologias ou técnicas convencionais. Para aplicações que requerem autonomia, mobilidade ou comportamentos sociais, aconselha-se o uso de técnicas que ofereçam ao desenvolvedor uma visão mais abstrata da aplicação, assim como do ambiente na qual ela está inserida. Neste contexto, a Inteligência Artificial provê o paradigma de *agentes inteligentes* que engloba um conjunto de metáforas, técnicas e ferramentas para facilitar a concepção, análise e implementação de aplicações que requerem algum tipo de comportamento inteligente. Os agentes são entidades de hardware e/ou de software naturalmente capazes de agir de forma autônoma e reativa, raciocinar pró-ativamente, se adaptar ao ambiente onde estão inseridos, assim como se comunicar, coordenar e cooperar com outras entidades.

1.5. OBJETIVOS

Para se dispor de agentes inteligentes vivendo em sociedade, deve-se garantir que eles tenham pelo menos dois componentes básicos: *raciocínio* e *comunicação* [07]. De fato, para garantir que os agentes se comportem de forma autônoma, estimulados por sensações externas (reatividade) e orientados a objetivos (pró-atividade), que tenham atitudes que os tornem adaptáveis aos ambientes (adaptabilidade) e que eles possam agir baseado em experiências

anteriores bem sucedidas (aprendizado), deve-se dotá-los de mecanismos de raciocínio.

Para as características sociais deve-se prover uma infraestrutura onde os agentes possam ser agregados em sociedades e trocar mensagens com outros agentes a fim de coordenadamente cooperar em prol da solução de problemas. Neste trabalho temos como objetivo a construção de um sistema que provê estas funcionalidades básicas na forma de APIs² para os agentes inteligentes que vivem em dispositivos móveis e em sociedade com outros agentes.

1.6. ABORDAGEM E CONTRIBUIÇÃO

Visto que já existem várias implementações dessas duas funcionalidades em J2SE (*Standard Edition*), a reutilização delas seria a forma mais rápida e prática de alcançarmos nossos objetivos. Portanto, foram escolhidos alguns sistemas para análise e, destes, selecionados dois (um para cada funcionalidade). Para cada sistema foram feitas adaptações e/ou extensões no intuito de adequá-los às limitações dos dispositivos e conseqüentemente, da linguagem de programação (J2ME). Buscamos por sistemas com uma arquitetura bem definida, com código aberto e que tivéssemos uma boa integração com os desenvolvedores para esclarecimentos e possíveis alterações dos ambientes originais.

A primeira API, denominada *KEOPS*, é uma adaptação da ferramenta *JEOPS (Java Embedded Objects Production System)* [08], um motor de inferência de primeira ordem com encadeamento progressivo que integra regras de produção e objetos Java, fruto de um trabalho de mestrado de um aluno do Centro de Informática da Universidade Federal de Pernambuco (UFPE).

A segunda API, denominada *KSACI* [09], é uma extensão da ferramenta *SACI (Simple Agent Communication Infrastructure)* [10], uma infra-estrutura de comunicação onde os agentes localizados em *PCs/Workstations* poderão ingressar em uma sociedade, enviar e

² *Application Programming Interfaces*

receber mensagens de outros agentes da mesma sociedade. SACI é uma ferramenta desenvolvida na Universidade de São Paulo (USP) por um aluno do curso de doutorado, cujas principais características são: uma arquitetura baseada em um modelo de sociedade de agentes e o completo suporte a KQML (Linguagem para Comunicação de Agentes) [11].

O *kAgent*, que é a junção dos dois sistemas acima, é um trabalho pioneiro a nível mundial. No início deste trabalho, nada havia sido implementado no tocante a desenvolvimento de agentes em J2ME. Esforços na construção de algumas ferramentas para comunicação de agentes para dispositivos móveis foram realizados em paralelo com o desenvolvimento do KSACI e publicados na conferência *Agents Theories, Architectures and Languages (ATAL)* em julho de 2001 [12][13]. Nenhum trabalho, no entanto, foi publicado no tocante a funcionalidades de raciocínio.

Acreditamos que com o desenvolvimento do *kAgent*, estaremos possibilitando um avanço no desenvolvimento de softwares baseados em agentes para a plataforma J2ME e iniciando um processo de criação de aplicações para dispositivos ubíquos, visto que, pouco tem sido feito nessa área até o presente momento.

1.7. PRÓXIMOS CAPÍTULOS

No capítulo 2, serão aprofundados os conceitos de agentes discutindo o porquê e como eles estarão inseridos nos dispositivos móveis. No capítulo 3 será detalhada a linguagem de programação J2ME com suas características, o processo de desenvolvimento de aplicações e a descrição de suas limitações quando comparada às versões anteriores da plataforma Java. No capítulo 5 serão introduzidos os conceitos e o estado da arte em termos de comunicação entre agente. O capítulo 6 detalhará o processo de desenvolvimento do KSACI com suas principais dificuldades e soluções, assim como ilustrará as principais características do KSACI usando um exemplo do seu funcionamento. No capítulo 7 serão introduzidos os conceitos e estado da arte com relação ao componente

de raciocínio, mais especificamente, será discutido como se embutir um mecanismo de inferência em dispositivos móveis. O capítulo 8 detalhará o processo de desenvolvimento do KEOPS com suas principais dificuldades e soluções. No capítulo 9 serão anunciadas as conclusões deste trabalho assim como enumerados alguns trabalhos futuros.

2.

AGENTES EM DISPOSITIVOS MÓVEIS

Com o aquecimento do mercado mundial de dispositivos móveis juntamente com o surgimento da rede sem fio de terceira geração e de J2ME, a idéia de que uma explosão de possíveis serviços ubíquos esteja para surgir está ficando cada vez mais clara [14]. O gráfico da Figura 2.1 mostra o crescimento dos rendimentos de serviços da rede 3G.

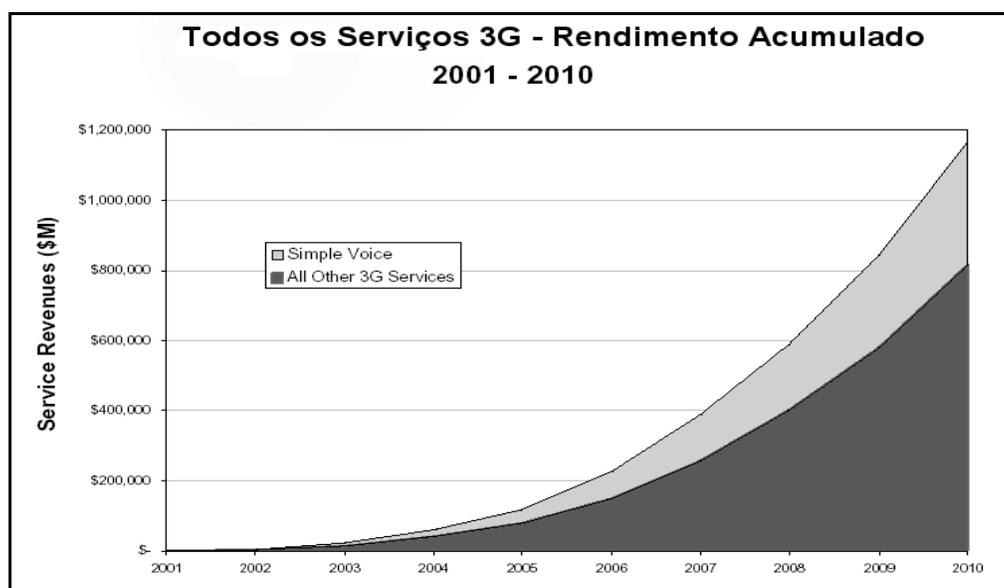


Figura 2.1: Crescimento do Rendimento dos Serviços 3G.
(Fonte: Telecompetitions Inc, Fevereiro de 2001)

De fato, o mundo está caminhando para um cenário onde os serviços computacionais poderão ser usados em qualquer lugar, a qualquer hora. Neste mundo de computação ubíqua, os sistemas estarão cada vez mais adaptados ao nosso modo de agir, contrariando o atual modo de interação homem-máquina onde nós temos que nos adaptar às formas limitadas de comunicação impostas pelos computadores sejam eles *desktops*, *notebooks*, *palmtops* ou celulares.

Neste contexto, diversos tipos de aplicações poderão surgir. Muitas destas aplicações complexas necessitam de mecanismos de autonomia, adaptabilidade e habilidades sociais. A abordagem dos agentes inteligentes, provinda da Inteligência Artificial, será de grande valia uma vez que os agentes são entidades naturalmente autônomas (o que inclui reatividade e pró-atividade) e capazes de se comunicar com outras entidades (humanas ou não).

Neste capítulo, será discutido como os agentes inteligentes estarão inseridos num mundo de computação ubíqua, dando ênfase aos requisitos deste tipo de computação, i.e., as características ou funcionalidades desejáveis aos agentes, assim como os requisitos computacionais para a sua implementação.

2.1. AMIGABILIDADE E INTELIGÊNCIA

A tecnologia envolvida na computação ubíqua deve atender a uma série de requisitos, sobretudo aqueles considerados do tipo, não-funcionais ou de qualidade, agrupados em três categorias [05]: *disponibilidade*, *confiabilidade* e *amigabilidade*.

Os requisitos de *disponibilidade* são os responsáveis pela localização dos serviços ubíquos, uma vez que estes serviços estarão localizados em todo lugar e a toda hora, como em um *shopping center*, onde serviços de anúncios dos estabelecimentos combinariam informações com os serviços de assistência pessoal dos potenciais clientes (possivelmente rodando em aparelhos celulares), para oferecerem produtos de acordo com o perfil do cliente. Serviços de transações comerciais poderiam iniciar processos de negociação entre

entidades representantes dos estabelecimentos e dos clientes, e fechar o negócio com a efetuação da transação e débito na conta do cliente.

A *confiabilidade* diz respeito ao quão segura e robusta é a aplicação. Ela torna-se cada vez mais crítica à medida que as organizações e pessoas dependem mais dos sistemas e aplicações computacionais distribuídos, conectados em rede com ou sem fio. Sistemas que precisam de tempos de respostas bastante reduzidos, os sistemas de tempo real, são exemplos claros de aplicações que necessitam de grande confiabilidade. Aplicações de transações bancárias, monitores de segurança eletrônica e serviços de telefonia são outros bons exemplos do nosso dia a dia.

O requisito de *amigabilidade* acaba sendo ajudado pelos dois últimos. De fato, para uma aplicação manter um alto grau de amigabilidade com seus usuários, provendo uma forma fácil e conveniente de interação dos indivíduos com os serviços ubíquos, devem-se garantir características de acessibilidade e confiabilidade. A forma convencional de interação homem-máquina não combina com o mundo da tecnologia ubíqua, que é vista como a utilização integrada de inúmeros tipos de mídias (texto, imagens, sons e vídeos) e a utilização de dispositivos ergonômicos, tais como luvas e óculos de realidade virtual, roupas com dispositivos inteligentes (*wearable computers*), etc.

Sistemas amigáveis demandam "*inteligência*", que se traduz na capacidade do sistema de minimizar a necessidade de interação com o usuário a fim de executar as tarefas desejadas. Por esta razão, é preciso dar mais autonomia aos sistemas de forma que, uma vez especificados os objetivos e perfis do usuário, eles sejam capazes de tomar decisões sem recorrer demasiadamente a intervenções humanas.

Para minimizar a demanda de fornecimento de informação pelo usuário, é imprescindível que os novos sistemas tenham também sensibilidade ao contexto, que disponham de um conhecimento *a priori* dos tipos de ações a serem requeridas e possam agir com autonomia. Um exemplo concreto da inteligência a serviço da amigabilidade é

quando um assistente pessoal é requisitado a comprar um determinado produto. De acordo com a localização e perfil do seu usuário, o assistente vasculha os estabelecimentos comerciais que possuem o melhor preço e negocia autonomamente com entidades representantes dos estabelecimentos para atingirem melhores preços, podendo até provocar um leilão inverso, isto é, verificando quem vende mais barato.

O desenvolvimento de tecnologias com estas características tem sido um dos temas de estudo da *Inteligência Artificial* (IA), em particular, das áreas de Agentes Inteligentes [15] e Aprendizado de Máquina [16]. A noção de *agentes* neste contexto é de grande valia, pois ela facilita a concepção de sistemas capazes de raciocinar e agir autonomamente, inserindo-se em um ambiente real (dinâmico, contínuo, não determinístico, etc.) [15]. De fato, uma abordagem baseada em agentes oferece um poderoso repertório de ferramentas, técnicas e metáforas que têm o potencial de melhorar consideravelmente a modelagem e implementação de vários tipos de software.

No entanto, apesar das pesquisas em modelos de agentes, restam vários desafios a serem superados. Em particular, dado que tais agentes estarão embutidos em sistemas de informação em geral, é fundamental que se estude e se proponha metodologias e ferramentas de programação orientada a agentes que se integrem com os paradigmas atuais, especialmente o de Orientação a Objetos. A integração entre programação declarativa e orientada a objetos evoca problemas teóricos e práticos que ainda não foram adequadamente tratados [08].

2.2. FUNCIONALIDADES DESEJÁVEIS

Os sistemas computacionais a serem implementados utilizando a abordagem de agentes inteligentes são construídos a partir de entidades de software, situadas em algum ambiente, capazes de realizar ações autônomas, tendo habilidades sociais e características adaptáveis.

Os agentes estão inseridos em um ambiente de onde eles recebem estímulos através de sensores, e onde realizam ações que o alteram de alguma forma. Dentre os ambientes onde os agentes podem estar inseridos incluem-se, por exemplo, um *shopping center* real, a *Internet*, sistemas de tráfego aéreo e jogos eletrônicos. Para estarem atuando num ambiente real como estes, os agentes necessitam de algumas características básicas tais como, autonomia, comportamentos sociais e adaptabilidade.

Autonomia é a capacidade do sistema de agir sem a direta intervenção humana (ou de outros agentes) e de ter mais controle sobre suas próprias ações e estado interno. A autonomia engloba dois conceitos bastante importantes: reatividade e a pró-atividade. Um agente *reativo* é aquele que percebe o ambiente onde está inserido e reage convenientemente a mudanças ocorridas nele. Os agentes *pró-ativos* são aqueles que exibem comportamento oportunista, direcionado a objetivos e tomam iniciativas quando apropriado e não agem simplesmente em resposta às mudanças ocorridas no ambiente. De fato, uma entidade autônoma não seria autônoma, se ela não reagisse às mudanças no ambiente e agisse sem rumos. Dentre os sistemas autônomos, podemos citar os reatores nucleares e os *demons* que monitoram ambientes reais e realizam ações que os modificam quando determinadas condições passam a ser verdade. Outro bom exemplo de autonomia se encontra no assistente pessoal citado na seção anterior onde ao pesquisar preços de artigos nas lojas de um shopping, o assistente autonomamente negocia com os agentes dos estabelecimentos em prol do melhor preço para o usuário.

Os agentes com comportamento *social* são os capazes de interagir, quando apropriado, com outros agentes humanos ou virtuais. Alguns padrões de interação entre agentes são: *cooperação* (trabalho agrupado e direcionado a um objetivo comum), *coordenação* (organização de atividades para a solução dos problemas que evita interações prejudiciais e explora as benéficas) e *negociação* (chegar a um acordo sobre o que é aceitável para todas as partes envolvidas). Dependendo da situação, estes conceitos podem ser interdependentes, por exemplo, para se negociar precisa-se que haja coordenação para explorar as iterações benéficas para ambas as

partes e de cooperação para que ambas as partes alcancem seus objetivos. O comportamento social fica bem exemplificado quando o assistente pessoal do exemplo anterior não tem habilidades suficientes para negociar. Para resolver este entrave, o assistente então procura por agentes que possuam as devidas habilidades, enviam-lhes mensagens para esclarecimentos ou delegação de tarefas. Assim, outra entidade coopera com o assistente objetivando a negociação do melhor preço.

Os agentes com características de *adaptabilidade* são agentes capazes de se adaptar às mudanças ocorridas no ambiente, isto é, capazes de agir baseados também em experiências prévias bem sucedidas e não só nas sensações do ambiente. No caso do assistente pessoal, a adaptabilidade estaria presente quando ele estivesse diante de uma técnica de negociação diferente das conhecidas anteriormente e mesmo assim o assistente conseguisse escolher uma boa tática para esta negociação baseada somente nas informações do ambiente e nos conhecimentos obtidos em negociações passadas.

2.3. REQUISITOS COMPUTACIONAIS

Visto os papéis dos agentes inteligentes em ambientes ubíquos, assim como suas habilidades (ou funcionalidades) desejáveis, faz-se necessário agora definir os requisitos computacionais para sua implementação, ou seja, que componentes computacionais básicos deve-se ter para que tais habilidades possam ser desenvolvidas.

Pode-se observar que para um agente estar situado em um ambiente, bastaria prover a ele mecanismos para percepção de estímulos e para atuação no ambiente. No entanto, para que um agente apresente os aspectos relacionados com a autonomia e adaptabilidade é necessário que se tenha um mecanismo de raciocínio. De forma análoga, para que os agentes, principalmente os que estão em outro contexto, possam usar os padrões de interação, é preciso disponibilizar um protocolo comum de conversação e uma infraestrutura.

2.3.1. COMPONENTE DE RACIOCÍNIO

Para se construir agentes como sistemas inteligentes e provê-los de características racionais, deve-se escolher a categoria de raciocínio a se trabalhar. Os principais tipos de raciocínios são: *dedução, abdução, indução e analogia* [15]. No caso do *raciocínio por dedução*, são utilizados fatos e regras para a produção de novos fatos. Por exemplo, suponha a seguinte regra para um lugar X:

[Regra]: *SE tem_fogo(X) ENTÃO tem_fumaça(X)*

Se a sentença *tem_fogo(X)* passa a ser verdade (passa a ser fato), então *tem_fumaça(X)* também passará. A dedução é a maneira mais fácil de se implementar e por isso a mais utilizada para modelar o conhecimento humano [08] e por isso a adotada neste trabalho.

Para a implementação de agentes com raciocínio dedutivo, dois componentes são necessários: um mecanismo de inferência, uma base de conhecimento e uma linguagem de manipulação que faz a junção deste conhecimento com a máquina de inferência. A base de conhecimento é formada por um conjunto de verdades sobre o mundo e o mecanismo de inferência ou máquina de inferência é responsável por inferir novos fatos a partir do conhecimento da base. A Figura 2.2 ilustra um agente situado em um determinado ambiente cujos sensores e efetadores alimentam e consultam uma base de conhecimento que, por sua vez, é gerenciada por uma máquina de inferência.

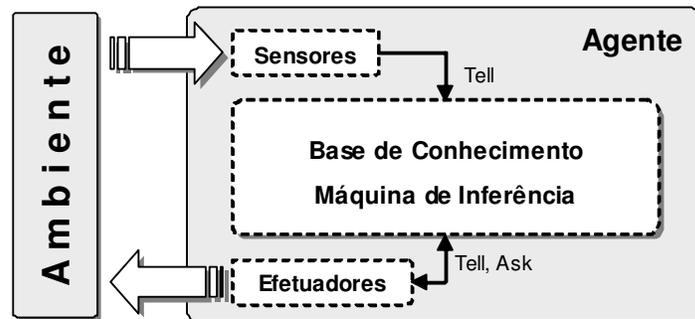


Figura 2.2: Arquitetura de raciocínio dedutivo.
Tell adiciona sentenças à BC. *Ask* consulta a BC.

Para se manipular o conhecimento usa-se geralmente linguagens de representação (LRC) que estão entre as linguagens de programação (precisas, porém não suficientemente expressivas) e as linguagens naturais (muito expressivas, porém ambíguas). Pode-se usar para representar o conhecimento, linguagens (predominantemente) declarativas (linguagens baseadas em lógica, regras ou objetos), linguagens procedimentais (Lisp, C, Pascal, etc), linguagens híbridas, linguagens de modelagem (UML [17]) e linguagens para consulta a banco de dados (SQL [18]).

Cada LRC tem pelo menos um mecanismo de inferência associado para manipular o conhecimento (raciocínio). Os provadores de teoremas trabalham com a lógica; a programação em lógica e os sistemas de produção trabalham com regras; as redes semânticas, os frames, a lógica descritiva e as linguagens de classe usam objetos; e assim por diante.

Dentre todos estes mecanismos de inferência, os sistemas de produção são os mais usados [19]. Eles utilizam as *regras de produção* como LRC, uma *memória de trabalho* onde os fatos são armazenados e um *algoritmo de encadeamento progressivo de regras* responsável por produzir novos fatos a partir de fatos conhecidos.

Como será descrita em capítulos posteriores, a linguagem de programação escolhida para implementação dos componentes, a J2ME, é orientada a objetos. A integração do componente de raciocínio (no nosso caso os sistemas de produção) com este tipo de linguagem é uma questão importante. Esta integração, chamada de EOOPS (*Embedded Object Oriented Production Systems*), tem sido constantemente motivo de estudo da comunidade de Inteligência Artificial e Engenharia de Software. O principal objetivo do EOOPS é a substituição de simples cadeias de caracteres que representam os fatos armazenados na memória de trabalho, por objetos. Esta substituição traz diversas vantagens não só no tocante a Engenharia de Software (*reusabilidade, modularidade, legibilidade, eficiência no desenvolvimento, manutenibilidade*), mas também na Engenharia do Conhecimento (representação natural de relações ontológicas).

2.3.2. COMPONENTE DE COMUNICAÇÃO

De forma semelhante à comunicação humana, a comunicação entre agentes depende de entendimentos prévios, como o contexto e as formalidades (ou protocolos) a serem seguidas nas diversas situações [11]. Uma interação eficiente é aquela que compartilha as informações e conhecimentos mutuamente entendidos. Esta interação requer quatro componentes fundamentais e distintos: uma linguagem comum, um entendimento comum do domínio, a habilidade de comunicar este conhecimento e uma infra-estrutura para esta comunicação.

Pode-se usar, entre outros, banco de dados, lógicas descritivas, linguagens de EOOPS, XML [20], RDF [21], KIF [22] e Prolog [23], UML [17], OWL [24], RuleML [25] e F-Logic [26] como linguagem de representação do conhecimento. Esta linguagem deve ser usada por ambos os agentes comunicantes, muito embora possam ser construídos tradutores de uma linguagem para outra.

Para o entendimento comum sobre o domínio da aplicação são comumente usados catálogos de classes etmológicas ou termos lógicos, relacionamentos e restrições entre eles, chamados *ontologias* [24]. Definida uma ontologia, ela pode ser distribuída para quem quiser desenvolver aplicações que abranjam o mesmo domínio.

Para comunicar o conhecimento costuma-se usar uma linguagem de comunicação de agentes (LCA). As LCAs são linguagens que representam os atos da fala (ou atos comunicativos), isto é, o objetivo comunicativo da mensagem a ser transmitida (ver capítulo 5).

Depois da definição da linguagem de comunicação, faz-se necessária a construção de uma infra-estrutura de comunicação que suporte esta linguagem. Esta infra-estrutura deve encapsular os detalhes de implementação dos mecanismos de rede (e.g. *TCP/IP* [27], *DCOM* [28], *CORBA* [29], *RMI* [30], e *SOAP* [31]) e prover funcionalidades básicas para a comunicação entre os agentes tais como, servidor de sociedades e facilitadores, funções para entrar e sair de sociedades, enviar mensagens, serviços de páginas brancas e amarelas, etc.

2.4. CONCLUSÃO

Neste capítulo foram discutidos os principais tópicos relacionados ao uso dos agentes nos dispositivos móveis. Vimos que a computação ubíqua requer certas características em suas aplicações, tais como, amigabilidade, disponibilidade e confiabilidade. Várias aplicações com estas características são facilmente resolvidas com a utilização dos agentes.

A implementação dos agentes em sistemas computacionais passa pela implementação de algumas funcionalidades básicas como autonomia, comportamentos sociais e adaptabilidade. Vimos também que estas funcionalidades seriam atendidas com a construção de dois componentes básicos: um para raciocínio e outro para comunicação. As principais características destes componentes também foram discutidas neste capítulo.



3. **LINGUAGEM DE PROGRAMAÇÃO**

Este capítulo discutirá a respeito das possíveis linguagens de programação para implementar os componentes básicos dos agentes que rodam em dispositivos móveis. No decorrer desta discussão, serão levantados alguns requisitos para uma linguagem de programação ideal, seguido das principais abordagens para se embutir agentes em dispositivos móveis. Baseado numa reflexão a respeito desses requisitos e das abordagens, a linguagem de programação escolhida será então detalhada.

3.1. REQUISITOS

Para se embutir agentes em dispositivos móveis, faz-se necessário o uso de uma infra-estrutura não só portátil, mas adequada. Ela deve prever e ter como base as restrições impostas pelos dispositivos (escassez de memória e poder de processamento), manter-se independente de plataforma, ter o comprometimento de boa parte dos fabricantes de dispositivos e possuir uma metodologia, processo, linguagem de modelagem e ferramentas de desenvolvimento no qual a linguagem pudesse ser facilmente integrada. No mais, seria interessante que a linguagem de programação a ser usada na

implementação dessa infra-estrutura já tivesse algum suporte ao desenvolvimento de agentes.

3.1.1. PORTABILIDADE

Devido à grande variedade de dispositivos móveis, a portabilidade é um requisito importante nessa área. Sem ela seria necessária a construção de diversas versões de uma aplicação, uma para cada tipo de dispositivo. O ideal é que se tenha uma infra-estrutura comum sobre o qual os desenvolvedores possam escrever aplicações sem se preocupar com detalhes de baixo nível, conceito este muito semelhante ao adotado por Java: *"Write once, run everywhere!"* (*"Escreva uma vez, rode em qualquer lugar!"*). É aconselhável, portanto que a linguagem escolhida para o desenvolvimento preveja a extensa gama de dispositivos e tenha o máximo de funcionalidade comuns a todos eles.

3.1.2. MEMÓRIA E PROCESSAMENTO

Os principais representantes dos dispositivos móveis aqui trabalhados são os celulares, *paggers* e *palmtops*. Estes tipos de dispositivos têm grandes restrições quanto ao poder de processamento e a memória. Os mais modernos aparelhos desse tipo possuem processadores com uma média de 33MHz, total de espaço de armazenamento de no máximo 8MB e um total de memória volátil (RAM) de no máximo 256KB.

A infra-estrutura para o uso de agentes num ambiente desse tipo deve levar em consideração todas estas restrições e prover a funcionalidade requerida de forma transparente.

3.1.3. COMPROMETIMENTO DA INDÚSTRIA

De nada andiantaria se a linguagem escolhida para a implementação não estivesse disponível nos dispositivos reais. Por este motivo, a linguagem a ser escolhida deve possuir o

comprometimento do maior número possível de fabricantes de dispositivos. Estes dispositivos devem estar aptos a executar aplicações produzidas na linguagem escolhida para a implementação.

3.1.4. FACILIDADES

Outro requisito importante é quanto às facilidades para o desenvolvimento das aplicações. A existência de uma metodologia, de um processo, de uma linguagem de modelagem e de ferramentas de desenvolvimento onde a linguagem escolhida seria naturalmente introduzida é um fator determinante para a escolha desta linguagem.

3.1.5. SUPORTE A AGENTES

O outro requisito adicional para a construção da infra-estrutura é o suporte ao desenvolvimento de agentes pela linguagem de programação, com a disponibilização de mecanismos que implementem as funcionalidades descritas em 2.2.

Este suporte pode estar disponível de duas formas diferentes: a linguagem de programação hospedeira possuindo componentes para o desenvolvimento de agentes em seu núcleo previamente implementados (*built-in*), ou esta mesma linguagem sendo utilizada para a implementação dos componentes na forma de APIs.

3.2. ABORDAGENS PARA EMBUTIR AGENTES NOS DISPOSITIVOS

Em se tratando de desenvolvimento de aplicações para dispositivos móveis em geral, duas abordagens podem ser vistas. Na primeira, as aplicações rodam fora do dispositivo e na segunda abordagem, dentro. Na primeira opção são usados sistemas clientes/servidores onde os clientes (agentes) localizados no dispositivo, trocam informações com o um servidor capaz de processar tarefas mais complexas. Esta abordagem é tipicamente usada pelas

aplicações desenvolvidas usando as tecnologias WAP e iMode que fazem uso de um *micro-browser* dentro do dispositivo para interpretar as informações vindas dos servidores.

No entanto, esta forma de desenvolver aplicações para dispositivos móveis requer uma conexão robusta e persistente o que às vezes não está disponível ou é muito cara para redes sem fio atuais. No mais, o raciocínio embutido é imprescindível para determinadas tarefas, requerendo reatividade em tempo real, tais como jogos interativos e assistentes de navegação para deficientes visuais.

Na segunda opção, as aplicações que rodam dentro do próprio aparelho podem ser programadas utilizando uma linguagem que faça uso de códigos nativos do sistema operacional hospedeiro ou utilizando uma outra linguagem que ofereça ao desenvolvedor uma abstração maior da plataforma com o uso de uma máquina virtual.

Neste contexto, esta seção revisa as principais linguagens de programação existentes em cada uma das abordagens, tendo em vista também os requisitos das aplicações para dispositivos móveis.

3.2.1. ACESSO NATIVO

Nesta categoria estão linguagens de programação que trabalham em conjunto com o sistema operacional hospedeiro, fazendo solicitações específicas a ele. As aplicações desenvolvidas seguindo esta abordagem geralmente são específicas para cada dispositivo. Os principais representantes desta categoria são as linguagens C e C++. Estas linguagens são geralmente usadas pelos fabricantes de dispositivos para desenvolver suas aplicações. As aplicações desenvolvidas em C estão geralmente muito ligadas ao sistema operacional, também escrito em C e, portanto, sem muita portabilidade.

3.2.2. MÁQUINA VIRTUAL

Nesta categoria estão linguagens de programação que trabalham sobre uma máquina virtual, uma camada de software que faz os acessos ao sistema operacional (ver Figura 3.1).

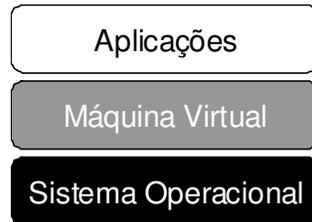


Figura 3.1: Uso de Máquina Virtual.

Esta máquina virtual disponibiliza ao desenvolvedor um conjunto de classes e interfaces que podem ser usadas para construção das aplicações. As aplicações desenvolvidas com esta abordagem têm um alto grau de portabilidade uma vez que este conjunto de classes e interfaces é comum a todos os tipos de plataformas. Os representantes dessa categoria são: *Qualcomm BREW* [33] e *J2ME* [05].

3.2.2.1. QUALCOMM BREW

BREW [33] é um ambiente desenvolvido pela QUALCOM [34], especialmente projetado para a construção de aplicações para dispositivos sem fio. Sua filosofia é bem parecida com a de Java, onde uma máquina virtual faz a ponte entre as aplicações e o sistema operacional hospedeiro. As diferenças ficam por conta da linguagem de programação utilizada para o desenvolvimento das aplicações em BREW, a linguagem C. No entanto o fabricante promete ter API para o desenvolvimento de aplicações em outras linguagens, inclusive Java. Outra diferença (e limitação), é que o BREW só roda em *chips* desenvolvidos pela própria QUALCOMM (*MSM 3100* ou mais recente) [35], isto é, BREW, além de fazer a ponte do sistema operacional e as aplicações, pode acessar diretamente as funções de um *chip* proprietário.

3.2.2.2. J2ME

A outra opção é o uso da tecnologia *Java 2 Micro Edition* - ou *J2ME* - o Java projetado especialmente para que os dispositivos móveis possam executar aplicações mais sofisticadas, que vão de jogos *online* a programas de uso corporativo. Com *J2ME*, pode-se desenvolver aplicações que rodam "dentro" dos dispositivos e não mais em um servidor (como em WAP).

O ponto alto de *Java 2 Micro Edition* é a portabilidade das aplicações desenvolvidas, uma vez que utiliza uma máquina virtual (a KVM). Diferentemente do que acontece no mercado de sistemas operacionais para *desktops*, a diversidade de programas para dispositivos móveis é quase tão grande quanto a de fabricantes. São mais de oitenta plataformas, que vão de *Windows CE*, *Symbian*, *Palm OS* a muitos outros sistemas proprietários que não chegam a ter nome comercial [36]. Por isso, o surgimento de um ambiente que consiga traduzir a mesma aplicação para qualquer ambiente deve trazer novos horizontes ao universo da computação ubíqua.

Seguindo os princípios que inspiraram a plataforma Java, a nova versão *Micro Edition* está pronta para interpretar qualquer sistema, independentemente de plataforma e tecnologia utilizadas. Isso significa que programas de vendas, controles de estoque, sistemas de pagamento ou mesmo jogos *online* poderão ser adaptados para executar em telefones celulares e *handhelds* de forma totalmente integrada às redes das empresas ou à Internet.

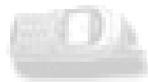
A tecnologia *J2ME* compreende a linguagem de programação Java, um conjunto de bibliotecas e uma máquina virtual responsável pela interpretação das aplicações. Tudo adaptado para operar com a mesma eficiência da tecnologia matriz (*J2SE*³ [37]), respeitando as limitações de hardware e poder de processamento dos dispositivos móveis.

³ *Java 2 Platform, Standard Edition*

3.3. CONCLUSÃO

Neste trabalho, adotamos J2ME que além de suas características de portabilidade e adequação aos requisitos dos dispositivos móveis, tem outras importantes características: não depende de nenhuma tecnologia ou fabricante, é baseada numa máquina virtual que garante a portabilidade das aplicações, é orientada a objetos (pois utiliza Java como linguagem de programação) e é fruto do consórcio de influentes fabricantes da área de telecomunicações (entre eles *Motorola*, *Nokia*, *Siemens* e *Ericson*) e da *Sun Microsystems* (desenvolvedora da linguagem Java). Outra importante vantagem de J2ME é que, apesar de não possuir nenhum componente *built-in* para o desenvolvimento de agentes, já existem diversas API (*Application Programming Interfaces*) com este propósito, produzidas por terceiros em J2SE ou J2EE e que podem ser estendidas ou adaptadas para J2ME.

Além disso, encontra-se em fase de consolidação, um programa entre *Motorola* (Brasil) e o *Centro de Estudos e Sistemas Avançados do Recife* (CESAR), colaborador constante do *Centro de Informática da UFPE* (CIn-UFPE). Este programa, nomeado *KJAVA Program*, tem por objetivo a formação de um grupo especializado no desenvolvimento e certificação de aplicações J2ME.



4.

J2ME

Esta seção trata dos principais aspectos da linguagem J2ME. Esta nova edição, com aproximadamente um ano de criação, é uma simplificação da linguagem J2SE realizada através da remoção e modificação de partes fundamentais de J2SE com o objetivo de criar um ambiente para construção de aplicações para dispositivos com restrições de memória e processamento. Inicialmente, é realizada uma apresentação histórica sobre o surgimento da linguagem e em seguida são mostrados alguns detalhes da arquitetura da linguagem.

4.1. HISTÓRICO

O surgimento de J2ME pode ser visto como uma retomada do objetivo inicialmente definido pela *Sun Microsystems* quando a linguagem Java foi concebida. Java tinha sido criada para executar em dispositivos pequenos, mas acabou sendo utilizada em computadores pessoais devido ao crescimento mercadológico dos mesmos. E devido a este novo mercado, as versões de Java começaram a aumentar de tamanho. A API foi sendo acrescida de mais e mais funcionalidades aumentando quase que 10 vezes o seu tamanho em bytes e 9 vezes o número de classes (ver Figura 4.1).

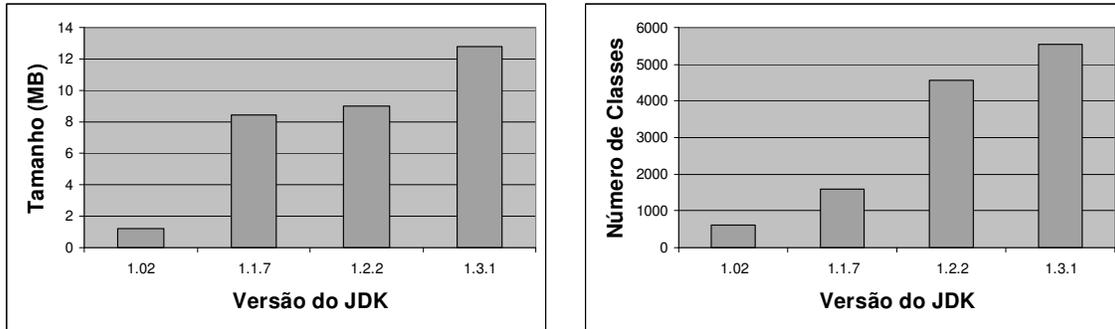


Figura 4.1: Crescimento no tamanho da linguagem Java

Assim, o mercado de dispositivos pequenos⁴ acabou sendo “esquecido” pela *Sun*, pois a utilização de Java tornou-se impossível devido ao seu tamanho. Com isso, estes dispositivos tiveram que voltar à sua forma original, geralmente privada, de execução e desenvolvimento de aplicações. De fato, isso é observado em celulares, *paggers*, rádios, receptores de TV via satélite e fornos microondas que vêm com um conjunto fixo de programas. Os usuários não podiam modificar o conjunto de softwares pré-existentes, instalar e desinstalar programas.

Houve uma tentativa da *Sun* de suprir esta deficiência com a criação de versões simplificadas de Java: *JavaCard* [38], *EmbeddedJava* [39] e *PersonalJava* [40]. Percebendo que ainda boa parte dos dispositivos existentes não eram atendidos, a *Sun* resolveu firmar um consórcio com as maiores empresas de aparelhos eletrônicos portáteis (entre elas *Motorola*, *Ericsson*, *Nokia*, *Sony* e *Siemens*) para chegar a uma solução. Dessa iniciativa, ficou decidido que a tradicional plataforma Java seria dividida em três edições, cada uma direcionada a certos setores do mercado: *Enterprise Edition* [41] (grandes aplicações comerciais), *Standard Edition* [37] (aplicações convencionais) e *Micro Edition* [05] (aplicações para dispositivos com limitações de processamento e memória). Realizada esta divisão, a *Sun* apresentou J2ME oficialmente durante a conferência *JavaOne* em 1999 [42].

⁴ Entenda-se por pequenos, dispositivos com grandes restrições de memória e processamento.

4.2. ARQUITETURA DE J2ME

A arquitetura de J2ME demorou quase um ano para ser definida porque deveria atender de forma irrestrita a todos os tipos de dispositivos com certas limitações de processamento e memória. Muitas das características da linguagem Java foram cortadas, pois não se adequavam a certos dispositivos, por exemplo, nem todo dispositivo usa tela gráfica como interface com o usuário ou realiza ligações telefônicas. Alguns dispositivos poderiam estar sobrecarregados com funcionalidades que jamais seriam usadas se características como estas fossem mantidas em J2ME.

J2ME adotou portanto uma arquitetura modular e escalável, como mostrado na Figura 4.2, que define três camadas de software construídas sobre o sistema operacional presente em cada dispositivo.



Figura 4.2: Modelo de Camadas de J2ME

As três camadas de software superiores serão apresentadas nas seções seguintes.

4.2.1. KVM (MÁQUINA VIRTUAL)

J2ME é uma tecnologia definida por várias partes que se complementam para prover um ambiente compatível com a grande quantidade de dispositivos. Como em todas as tecnologias Java, na base de todas estas partes está uma máquina virtual com objetivo de garantir a portabilidade e execução de programas em qualquer dispositivo. A *portabilidade* é garantida desenvolvendo-se esta

máquina virtual para cada dispositivo existente, levando em conta o sistema operacional presente e as características dos aparelhos. A execução de programas nestes dispositivos simples resultou num grande desafio de otimização para tornar esta máquina virtual o mais simples possível.

Diferentemente do termo JVM (*Java Virtual Machine*), utilizado para referenciar as máquinas virtuais, em J2ME a máquina virtual é chamada de KVM (*Kilobyte Virtual Machine*), em referência ao seu pequeno tamanho, usualmente menor que 128KB. A KVM não é apenas pequena, ela é uma implementação bastante otimizada da máquina virtual Java considerando todas as restrições de memória e processamento envolvidas nos pequenos aparelhos.

Os recursos oferecidos por J2ME dependem de qual configuração e perfil são utilizados. Sendo assim, a própria máquina virtual também varia de acordo com os requisitos presentes em cada ambiente formado.

4.2.2. CONFIGURAÇÃO

A *camada de configuração* define os recursos Java que fazem parte de uma larga categoria de dispositivos que compartilham certas características importantes, tais como capacidade de memória, comunicação, consumo de energia e interface com o usuário. Em outras palavras, uma configuração define o "menor denominador comum" dos recursos da plataforma Java que serão suportados em todos os dispositivos pertencentes à larga categoria considerada pela configuração.

Para evitar uma grande segmentação, apenas duas configurações existem, *Connected Device Configuration* - CDC [43] e *Connected Limited Device Configuration* - CLDC [44]. Estas configurações foram definidas a partir de fóruns de discussão com centenas de empresas, desenvolvedores e usuários de dispositivos que se mostraram interessados na definição destas configurações. A seguir seguem as suas principais características.

4.2.2.1. CLDC

A configuração CLDC é direcionada para dispositivos com grandes restrições de memória e processamento. Nesta categoria estão dispositivos como os celulares, *paggers* e os PDAs mais simples, cujas características comuns são:

- Memória disponível para J2ME entre 128KB e 512KB;
- Baixa qualidade e largura da banda de conexão, em torno de 9600 bps e intermitente, por ser sem fio;
- Grandes restrições de energia por ser fornecida através de baterias;
- Interface com o usuário muito limitada em tamanho e resolução;
- Processador de 16 ou 32 bits e frequência em torno de 25 MHz.

Baseadas nestas restrições, um conjunto de quatro pacotes Java foi definido para contemplar os recursos igualmente suportados por todos os dispositivos abrangidos pela configuração CLDC - *java.lang*, *java.util*, *java.io* e *javax.microedition*. Com exceção do pacote *microedition*, os outros pacotes são subconjuntos dos pacotes equivalentes em J2SE. No entanto, todos foram totalmente redefinidos para remover classes desnecessárias, bem como métodos e atributos [45]. Os principais recursos oferecidos por todos estes pacotes são os seguintes:

- Suporte a tipos primitivos de dados, tais como, *Boolean*, *Byte*, *String* e *Integer*;
- Suporte a algumas estruturas de dados, tais como, *Hashtable* e *Vector*;
- Suporte a importantes recursos da linguagem J2SE, tais como utilização de *Threads*, *Exceptions* e *Garbage Collector*;
- Definição de um *framework* básico para comunicação de dados.

4.2.2.2. CDC

A configuração CDC tem o papel de cobrir os dispositivos com recursos superiores aos dispositivos tratados pela configuração CLDC, porém inferiores aos encontrados nos computadores pessoais, que já contam com o próprio J2SE. Nesta categoria estão dispositivos como os televisores, computadores de bordo/sistemas de navegação de carros e *screenphones*, cujas características comuns são as seguintes:

- Memória disponível para Java entre 2MB e 16MB;
- Conectividade de alta qualidade e largura de banda;
- Dispositivos fixos e com boa alimentação de energia;

De forma análoga à configuração CLDC, um conjunto de pacotes Java foi definido para contemplar os recursos igualmente suportados por todos os dispositivos abrangidos por CDC - *java.lang*, *java.util*, *java.net*, *java.io*, *java.text*, *java.security* e *javax.microedition*. Todas as classes presentes nesses pacotes são idênticas (com as devidas adaptações) às existentes em J2SE nos pacotes equivalentes. Estes pacotes também englobam todos os presentes em CLDC, com o objetivo de manter a compatibilidade entre as duas configurações.

A Figura 4.3 ilustra a relação de pertinência entre os recursos de CLDC, CDC e J2SE. Observar que tanto CDC quanto CLDC contém recursos adicionais à J2SE, como o pacote *javax.microedition*, por exemplo.

Devido a esta existência de duas configurações, duas máquinas virtuais diferentes foram especificadas com o objetivo de suportar as características presentes em cada configuração. Uma delas é a KVM já comentada, que é voltada para a configuração CLDC. A outra, chamada de CVM (de *Compact Virtual Machine*) [43] é direcionada para a configuração CDC.

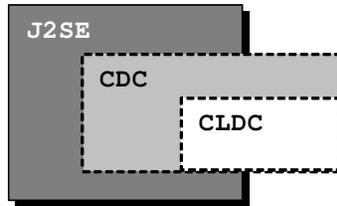


Figura 4.3: Cobertura de classes de CDC e CLDC em relação à J2SE

4.2.3. PERFIL (PROFILE)

O objetivo da camada de perfil é especializar uma determinada configuração para uma parte do mercado largo que ela atinge. Cada perfil define a API voltada para demandas específicas de um segmento de mercado, tais como, carros, máquinas de lavar, celulares e televisores.

Um perfil é a camada mais próxima para os usuários e desenvolvedores de aplicações. Ela provê mecanismos como entrada de dados, interface com o usuário, persistência de dados e conectividade da forma ideal para o segmento de mercado abrangido pelo perfil.

Naturalmente, um programa escrito para um dado perfil tem a portabilidade garantida para qualquer dispositivo que suporte tal perfil. A idéia é que isso ocorra da mesma forma que aplicações para PCs são desenvolvidas para um determinado sistema operacional. Não há preocupações a respeito do fabricante da máquina que irá executá-lo.

O número de perfis é bem maior que o de configurações por lidar com segmentos específicos de mercado. Atualmente existem dois perfis definidos, *Mobile Information Device Profile* - MIDP [46] para a configuração CLDC, e *Foundation Profile* [47] para a configuração CDC. Vários outros perfis estão em processo de desenvolvimento, também através de fóruns com toda a indústria interessada. Entre eles destaca-se o perfil *PDA Profile* [48] para a configuração CLDC que é voltado para as demandas específicas dos PDAs. A seguir seguem maiores detalhes sobre o MIDP e *Foundation Profile*.

4.2.3.1. MIDP

Em função da grande exigência da indústria de telefonia móvel, o primeiro perfil da tecnologia J2ME a ser desenvolvido foi o MIDP, voltado para dispositivos móveis como os celulares e *paggers*. A sua especificação foi realizada em conjunto com mais de 20 empresas da indústria dos dispositivos móveis, resultando em um conjunto de pacotes Java que suplementam a configuração CLDC com diversos novos recursos, destacando-se:

- Definição de classes para criação da interface gráfica dos programas;
- Suporte à persistência de dados pelas aplicações, através de um pequeno banco de dados baseado em registros que é gerenciado pelo próprio MIDP;
- Implementação do protocolo HTTP para comunicação remota de dados utilizando a própria rede sem fio dos equipamentos;
- Representação da entidade aplicação, nomeada *MIDlet*, com um comportamento muito similar aos *Applets* utilizados em J2SE, que deve ser utilizada pela aplicação para que possa ser executada pela KVM.

O perfil MIDP alcançou grande sucesso. Primeiramente pelo fato da *Sun* ter liberado uma ferramenta de desenvolvimento gratuita para CLDC/MIDP, chamada de *Sun J2ME Wireless Toolkit* [49], que permitiu a emulação e desenvolvimento de programas J2ME em PCs, uma vez que os dispositivos ainda não tinham suporte a J2ME. Segundo, ainda mais importante, foi a grande aderência mercadológica a este padrão pela indústria de telefonia móvel. O sucesso foi tanto, que a *Sun* realizou uma implementação da KVM para o sistema operacional *PalmOS*, utilizado nos *Palms*, para dar suporte à CLDC e MIDP nesses PDAs [50]. Com isso, já é possível utilizar J2ME em *Palms* gratuitamente mesmo sem o perfil específico para PDAs estar totalmente finalizado, bastando para isso instalar esta KVM disponibilizada para *download* pela *Sun*.

Apesar do suporte proporcionado pelo conjunto CLDC/MIDP, alguns recursos bastante importantes não são encontrados. Entre estas ausências destacam-se:

- Ausência de ponto flutuante, devido ao fato de operações envolvendo números reais serem muito caras para os processadores encontrados nos dispositivos móveis cobertos pela plataforma;
- Ausência da reflexão Java, o que implica que programas não podem inspecionar o conteúdo de classes, objetos e métodos, para, por exemplo, realizar serialização de objetos e execução remota de métodos (RMI) [51];

4.2.3.2. FOUNDATION PROFILE

O *Foundation Profile* [47], também definido em conjunto com a indústria interessada, especifica um conjunto de pacotes Java que suplementam a configuração CDC com os principais pacotes ausentes nessa configuração, destacando-se o suporte a *sockets*, internacionalização e a implementação das classes ausentes nos pacotes *java.lang* e *java.io*.

O objetivo da Sun com o perfil *Foundation* é servir como base para outros perfis mais avançados do que ser de fato utilizado isoladamente para o desenvolvimento de programas [52]. Nesse sentido, atualmente existem alguns perfis bem interessantes em fase de especificação que poderão ser utilizados sobre o *Foundation Profile*. Um deles é chamado *Personal Profile* [53], e tem como objetivo, prover um ambiente similar ao oferecido pela tecnologia *Personal Java*.

A Figura 4.4 resume a atual situação da arquitetura geral da linguagem Java, ilustrando também com mais detalhes as máquinas virtuais, configurações e perfis de J2ME.

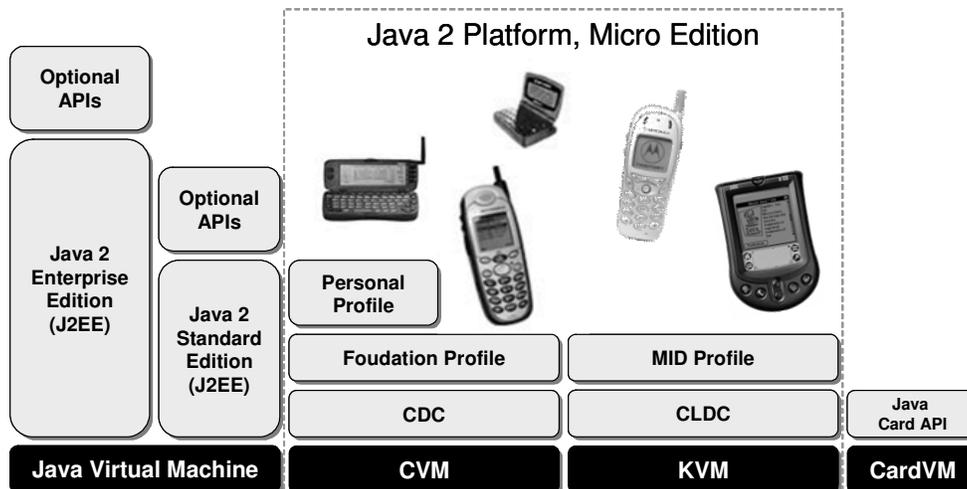


Figura 4.4: Atual arquitetura de Java [45]⁵

4.3. PROCESSO DE DISTRIBUIÇÃO DAS APLICAÇÕES

Devido às limitações dos dispositivos, alguns passos para a distribuição de aplicações tiveram de ser alterados. Após o desenvolvimento normal das classes da aplicação (código fonte), deve-se executar uma compilação Java convencional do código (utilizando o ambiente de desenvolvimento J2SE) seguido de uma pré-verificação das classes compiladas. Em J2SE, o processo de verificação de classes é um processo realizado pela JVM em tempo de execução. Devido à complexidade das operações de verificação de classes e aos escassos recursos dos dispositivos, este processo teve que ser dividido, sendo parte realizado pela JVM, que convencionou-se chamar de *pré-verificação*, e parte realizado pela KVM (ver Figura 4.5).

Depois de compiladas e pré-verificadas, as classes são compactadas em formato JAR e um descritor é gerado com informações sobre a aplicação. Este descritor (arquivo JAD - *Java Application Descriptor*) juntamente com o arquivo JAR gerado, serão usados no processo de *download* e instalação da aplicação.

⁵ Java Card [97] é uma API Java para smart cards e outros dispositivos com severas limitações de memória.

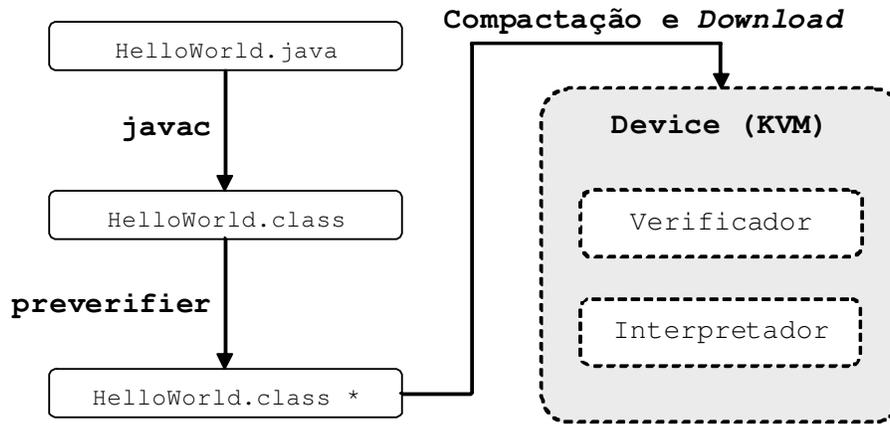


Figura 4.5: Processo de distribuição de uma aplicação

4.4. PROCESSO DE INSTALAÇÃO DAS APLICAÇÕES

Uma funcionalidade essencial de todos os ambientes de desenvolvimento para J2ME é a capacidade de geração de arquivos binários com todo o resultado da compilação de um programa, ou seja, o arquivo que precisa ser transferido do emulador para ser instalado e executado no dispositivo real. O conteúdo desse arquivo compactado em formato JAR (*Java Archive*) é formado pelo resultado da compilação de todas as classes e recursos da aplicação em desenvolvimento. O arquivo JAD mencionado anteriormente contém informações sobre o projeto, tais como o nome do *MIDlet*, o tamanho do arquivo JAR, a versão da aplicação e versões utilizadas da configuração e perfil.

Para que a instalação desses programas nos dispositivos ocorra, os dispositivos devem implementar um mecanismo chamado *Java Application Manager (JAM)*, cuja especificação é definida pelo perfil J2ME em uso pelo dispositivo. Um cenário típico de instalação de um programa é mostrado na Figura 4.6 [54]. Nesta situação, um usuário de um celular com suporte a MIDP acessa uma página na Internet que mostra os nomes de um conjunto de aplicações disponíveis para compra. Quando o usuário seleciona uma destas através do aparelho, automaticamente é feito o *download* de um arquivo texto que descreve o conteúdo do programa (o arquivo JAD) utilizando a rede sem fio do

dispositivo. Uma vez que o JAM tenha checado a possibilidade de instalação do programa através das informações do arquivo descritor, o processo de *download* da aplicação é realizado (arquivo JAR). O JAM irá salvar, instalar a aplicação no dispositivo e adicioná-la à lista de programas instalados pelo usuário.

O modelo atual de uso do JAM impede que novas classes sejam instaladas para um dado programa presente no dispositivo. Devido a esta limitação, se uma atualização de um programa for necessária, o *download* e instalação de todo o novo programa (novo arquivo JAR) é necessário.

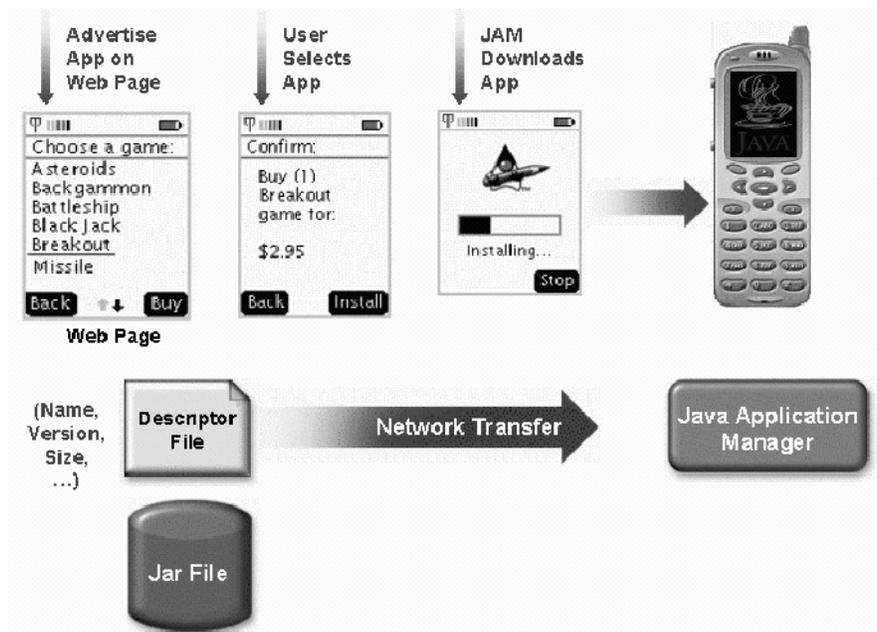


Figura 4.6: Processo de instalação de programas em dispositivos

Este mesmo processo de instalação também pode ser feito via um cabo (serial ou USB) ligando o PC ao dispositivo. O processo de instalação é semelhante, com a diferença que o *download* é feito via o próprio cabo de conexão. Nesse caso, é necessária a utilização de um software para PCs fornecido pelo fabricante do dispositivo, bem como a compra do cabo de conexão.

4.5. ADERÊNCIA MERCADOLÓGICA

Com a definição da arquitetura de J2ME para dispositivos móveis (CLDC/MIDP), diversos fabricantes iniciaram imediatamente o desenvolvimento das KVMs para os seus sistemas operacionais, de forma que hoje, com pouco mais de um ano, diversos fabricantes de dispositivos móveis já oferecem, ou anunciam, dispositivos com suporte à J2ME. Empresas como Siemens, Nokia, NEC, Samsung, Motorola, Sony, Toshiba, Casio e Sharp já desenvolveram dispositivos com a tecnologia J2ME. O número de modelos de celulares já ultrapassa 30, e este número não para de crescer [55]. Só a Nokia anunciou que irá fabricar em 2002, 50 milhões de aparelhos com suporte a J2ME, e que este número será o dobro em 2003 [56]. Na Figura 4.7, alguns modelos de celulares que já suportam J2ME são ilustrados.



Figura 4.7: Exemplos de dispositivos que suportam J2ME

A produção de aplicações para dispositivos móveis promete ser um grande nicho de mercado nessa nova tecnologia, além de um campo aberto à pesquisas, visto que não existem soluções para muitos problemas, como, por exemplo, a falta de suporte de J2ME a requisitos importantes para a implementação de abordagens baseadas nos agentes inteligentes provenientes da Inteligência Artificial.

Esta situação, como era de se esperar para uma tecnologia em seus primeiros anos de vida, traz grandes desafios, que serão explorados no decorrer deste trabalho.

4.6. CONCLUSÕES

Como ilustrado ao longo deste capítulo, J2ME é resultado de um longo processo de evolução da *Sun* em busca de uma tecnologia capaz de permitir o desenvolvimento em massa de aplicativos voltados para dispositivos móveis. Além da investida tecnológica da *Sun*, a grande aceitação que J2ME tem alcançado entre os fabricantes de dispositivos e empresas da área mostra que esta tecnologia tem tudo para obter um grande sucesso.

No decorrer deste trabalho, quando fizermos alusão a J2ME estaremos nos referenciando ao par CLDC 1.0 / MIDP 1.0, uma vez que os dispositivos abordados por este conjunto são o alvo principal deste trabalho.



5. **COMUNICAÇÃO**

Os sistemas modernos de computação freqüentemente envolvem múltiplos tipos de equipamentos interagindo de forma distribuída [57]. Num mundo de tecnologias ubíquas, não só os dispositivos serão variados, mas também as aplicações. Neste sentido, a habilidade de comunicação é um importante atributo que os agentes devem possuir. Ambientes que contêm mais de um agente geralmente exigem o intercâmbio de informações, o que torna claro a necessidade de uma linguagem de comunicação comum. Apesar desta afirmação ser bem aceita pela comunidade de pesquisadores, o problema é definir qual a linguagem ideal.

Neste capítulo serão apresentados os requisitos para uma linguagem de comunicação entre agentes, assim como as iniciativas que produziram especificações para as duas principais linguagens de comunicação atuais. Ao final do capítulo serão descritas algumas implementações bem sucedidas de infra-estruturas para comunicação, entre elas, a ferramenta escolhida para a implementação do KSACI.

5.1. LINGUAGENS DE COMUNICAÇÃO

As Linguagens de Comunicação normalmente são baseadas na teoria dos atos da fala (ver Apêndice A). Na Teoria dos Atos da Fala as mensagens são tratadas como ações (ou atos comunicativos) desde que

elas sejam intencionadas para executarem alguma ação [57]. As Linguagens para Comunicação de Agentes (LCA) são mais do que um protocolo para troca de dados, uma vez que a atitude sobre a informação trocada pelos agentes também é comunicada. Uma LCA pode ser pensada como um protocolo de comunicação, ou uma coleção de protocolos, que suporta muitos tipos de mensagens [57].

5.2. REQUISITOS

De maneira geral, uma LCAs possuem os seguintes requisitos: *Forma, Conteúdo, Semântica, Implementação, Rede, Ambiente e Segurança* [57].

Quanto à sua *forma*, a linguagem de comunicação entre agentes deve ser declarativa, sintaticamente simples, legível por humanos, concisa, fácil de analisar e de ser traduzida.

Quanto ao *conteúdo*, a linguagem deve ser estruturada em camadas, de forma a ser usada em outros sistemas. Esta divisão em camadas facilita a integração das linguagens com as aplicações enquanto provêem uma estruturação para a compreensão da linguagem. Em particular, uma distinção deve ser feita entre linguagem de comunicação que expressa *atos comunicativos* (a própria LCA) e a linguagem de contexto (ou de conteúdo) que expressa *fatos sobre o domínio* (que é utilizada dentro da LCA).

A *semântica* de uma linguagem de comunicação deve apresentar algumas propriedades comumente esperadas na semântica de qualquer outra linguagem [57]. Por exemplo, ela deve ser fundamentada em uma teoria e não ter ambigüidades, ter uma forma canônica (similaridade entre o significado e representação) e distinguir cuidadosamente a localização e o tempo, uma vez que a linguagem de comunicação é projetada para interações prolongadas entre aplicações dispersas espacialmente.

A *implementação* de uma LCA deve idealmente apresentar algumas características especiais, tais como: eficiência adequada,

concordância com as tecnologias de software existentes, ter uma interface amigável, ser transparente ao usuário, ser independente de plataformas e linguagens de programação e ser flexível.

Os requisitos de rede de uma LCA devem ter uma boa adequação com as modernas tecnologias de rede uma vez que a comunicação entre agentes utiliza conceitos envolvendo a comunicação em rede. A LCA deve suportar os tipos básicos de comunicação (*ponto-a-ponto*, *multicast* e *broadcast*) assim como, as conexões síncronas e assíncronas. Os protocolos de comunicação usados pelas LCAs devem ser independentes do mecanismo de transporte utilizado (e.g. *TCP/IP*, *SMTP (email)*, *HTTP*)

Quanto ao ambiente usado pelos agentes, os principais requisitos são a distributividade, a heterogeneidade, a dinamicidade, suporte à interoperabilidade com outras linguagens e protocolos, e a facilidade de anexação a outros sistemas.

Uma LCA deve oferecer além da *confiança* e *segurança*, uma maneira de garantir a autenticidade de agentes suportando mecanismos razoáveis para identificação e sinalização de erros e advertências.

Para a especificação das linguagens de comunicação baseadas nestes requisitos, surgiram algumas iniciativas. O primeiro esforço neste sentido foi o KSE (*Knowledge Sharing Effort*) do Departamento de Defesa Americano (DARPA). Na segunda iniciativa, a FIPA (*Foundation for Intelligence Physical Agent*), foram conservadas as vantagens e solucionadas as desvantagens da linguagem de comunicação definida pelo KSE: a KQML. As próximas duas seções descreverão com detalhes as duas iniciativas: KSE e FIPA.

5.3. KSE

KSE é a sigla para *Knowledge Sharing Effort*, ou esforço para o compartilhamento de conhecimento. KSE é uma iniciativa da agência do departamento de defesa americano DARPA (*Defense Advanced Research Projects Agency*) que contou com a participação de pesquisadores da

academia e da indústria [58]. O objetivo do KSE é a definição de tecnologias, metodologias e ferramentas de software para o compartilhamento e reuso de conhecimento em tempo de projeto, implementação e execução.

KSE usa a abordagem declarativa que é baseada na idéia de que a comunicação pode ser modelada com a troca de afirmações declarativas, sendo ao mesmo tempo compactada e suficientemente expressiva para comunicar uma grande quantidade de tipos de informações. Este grupo definiu uma Linguagem de Comunicação de Agentes (LCA) como sendo uma linguagem que possuísse os seguintes componentes: um vocabulário comum, uma linguagem (lógica) interna e uma linguagem externa.

5.3.1. ONTOLOGIA

No processo de conversação, o primeiro requisito é o entendimento do assunto em questão. As ontologias se propõem a prover este entendimento possibilitando a comunicação independente de tecnologias e arquiteturas, entre sistemas de software. Em termos conceituais, uma ontologia é uma especificação de uma conceitualização, ou seja, uma ontologia é uma descrição (como uma especificação formal de um programa) dos conceitos e dos relacionamentos que podem existir para um agente ou para uma comunidade de agentes atuando em determinado domínio [24].

Os principais componentes das ontologias são: um vocabulário de termos básicos, uma especificação precisa do que estes termos significam e uma especificação dos relacionamentos entre eles. Desta forma, pode-se assegurar que dois agentes estejam utilizando os mesmos termos durante o processo de comunicação. Em outras palavras, a utilização de uma ontologia permite a definição de um contexto único, eliminando-se a ambigüidade. Cada termo na ontologia tem uma descrição escrita em linguagem natural para ser usada por humanos no entendimento da mesma. Cada termo tem também uma anotação formal para serem usadas pelos agentes.

A proposta do KSE para uma ontologia é a *Ontolingua* [59], uma linguagem para descrição de ontologias e um conjunto de ferramentas de suporte. Mais recentemente o DARPA definiu junto com o W3C [61] alguns possíveis sucessores da Ontolingua baseados em XML e RDF: DAML [69], DAML-OIL [70], OWL [24].

5.3.2. LINGUAGEM INTERNA

A *linguagem interna*, também chamada de *linguagem de conteúdo*, é onde a parte lógica da mensagem é composta (usando o vocabulário comum, os termos da ontologia escolhida). É ela que manipula o conhecimento armazenado nos agentes.

A proposta do KSE para uma *linguagem interna* foi batizada de KIF (*Knowledge Interchange Formalism*). KIF é uma linguagem lógica projetada para servir como *interlingua*, ou seja, uma linguagem que atua como mediador em traduções entre linguagens. KIF é uma versão pré-fixada do cálculo de predicados de primeira-ordem com extensões para o suporte de meta-operadores e definições. Sua descrição inclui uma especificação para sua sintaxe e outra para sua semântica.

Por exemplo, as sentenças abaixo inserem três registros em um banco de dados de funcionários (os argumentos significam respectivamente o identificador do funcionário, o seu departamento e seu salário).

```
(salary 015-46-3946 widgets 72000)
(salary 026-40-9152 grommets 36000)
(salary 415-32-4707 fidgets 42000)
```

Informações mais complexas podem ser expressas através do uso de termos complexos, por exemplo, a sentença seguinte afirma que um navio é maior que um outro:

```
(> (* (width navio1) (length navio1))
    (* (width navio2) (length navio2)))
```

A expressão abaixo é um exemplo de uma sentença complexa em KIF que utiliza operadores lógicos para ajudar na codificação de informações lógicas, tais como negação, disjunção e regras. Ela

declara que é positiva a exponenciação de qualquer número real (?x) à potência de um número par (?n)

```
(=> (and (real-number ?x) (even-number ?n))
      (> (expt ?x ?n) 0)
)
```

5.3.3. LINGUAGEM EXTERNA

A *linguagem externa* é a linguagem que além de representar a atitude proposicional das mensagens enviadas, encapsula as principais características da mensagem, tais como o conteúdo (escrito na linguagem interna), a ontologia usada, a identificação dos agentes comunicantes e os identificadores relacionados à conversação em si.

A proposta do KSE para uma *linguagem externa* é a KQML (*Knowledge Query and Manipulation Language*). A linguagem KQML é tanto uma linguagem como um conjunto de protocolos para comunicação de agente que não depende nem de ontologia nem de linguagem interna específicas.

5.3.4. KQML

KQML é uma linguagem dividida em três camadas como mostra a Figura 5.1 abaixo:

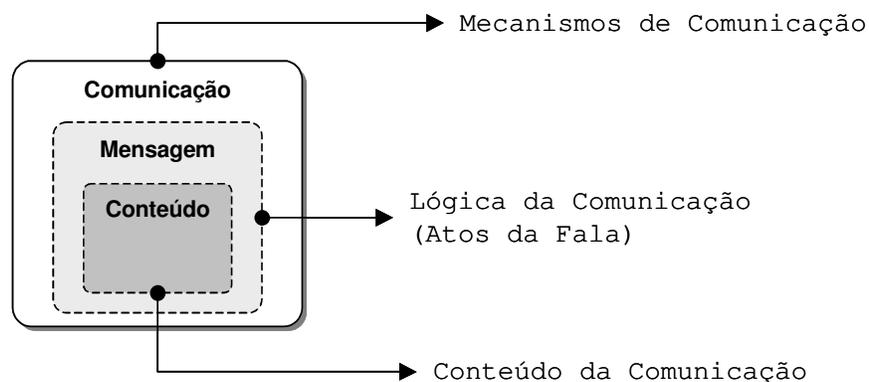


Figura 5.1: Camadas da linguagem KQML

A *camada de conteúdo* possui a mensagem e pode ter no seu conteúdo qualquer representação de linguagem.

A *camada de mensagem* é usada para codificar a mensagem a ser transmitida de uma aplicação para outra. A função primária desta camada é identificar o protocolo usado para transmitir a mensagem e a primitiva que o transmissor anexa ao conteúdo (linguagem, ontologia e tipos de atos de fala). A primitiva pode ser uma declaração, uma consulta, um comando ou um conjunto de primitivas conhecidas.

A *camada de comunicação* engloba um conjunto de características para aceitar parâmetros de alto nível, tais como a identidade de quem envia e recebe a mensagem e um identificador único associado com a mensagem.

KQML foi projetada para ser usada com vários mecanismos (atualmente há implementações que usam *TCP/IP*, *SMTP (email)*, *HTTP* e *CORBA*). O agente que usa KQML pode falar diretamente com outro agente ou pode enviar mensagens a múltiplos agentes do mesmo grupo. O ambiente operacional para agentes KQML é altamente distribuído, heterogêneo e extremamente dinâmico. Para satisfazer as exigências de tal ambiente, a linguagem KQML provê ferramentas formais que trabalham com outras linguagens e protocolos.

5.3.4.1. SEMÂNTICA DAS MENSAGENS KQML

Baseadas na teoria dos atos de fala (ver Apêndice A), as categorias básicas de primitivas (ou performativa, ou ainda ato comunicativo) são as seguintes:

Categoria	Primitivas
Serviços de informação	<i>tell, deny, achieve, cancel, untell,...</i>
Consultas básicas	<i>ask-if, ask-one, ask-all, reply, ...</i>
Banco de dados	<i>insert, delete, delete-one, delete-all,...</i>
Respostas básicas	<i>error, sorry, reply, ...</i>

Múltiplas respostas para consultas	<i>stream-about, stream-all, eos,...</i>
Geradoras	<i>standby, ready, next, discard, generator,..</i>
Capacidade e notificação	<i>advertise, subscribe, import, export,...</i>
Rede	<i>register, forward, broadcast, route,....</i>

Tabela 5.1: Categorias básicas de primitivas KQML

Embora haja um conjunto predefinido de primitivas reservadas, ele não é requerido nem no mínimo, nem no máximo, ou seja, um agente que usa KQML pode escolher manipular apenas poucas primitivas (talvez uma ou duas) ou escolher usar primitivas adicionais se eles concordarem na sua interpretação e no protocolo associado a cada uma [11].

KQML também possui um conjunto de *parâmetros-chave* reservados, sendo útil para estabelecer graus de uniformidade dos parâmetros e para que programas entendam primitivas desconhecidas. Os parâmetros reservados e seus significados estão apresentados na tabela seguinte:

Parâmetro	Significado
<i>performativa</i>	<i>Ato comunicativo da mensagem</i>
<i>:content</i>	<i>A informação sobre qual primitiva expressa uma atitude</i>
<i>:in-reply-to</i>	<i>O rótulo esperado em resposta</i>
<i>:language</i>	<i>O nome de uma linguagem utilizada no conteúdo</i>
<i>:ontology</i>	<i>O nome da ontologia utilizada no conteúdo</i>
<i>:receiver</i>	<i>O receptor da mensagem</i>
<i>:reply-with</i>	<i>Um rótulo para a (possível) resposta</i>
<i>:sender</i>	<i>O transmissor da mensagem</i>

Tabela 5.2: Parâmetros reservados de KQML

A figura seguinte resume os principais elementos estruturados de uma mensagem KQML.

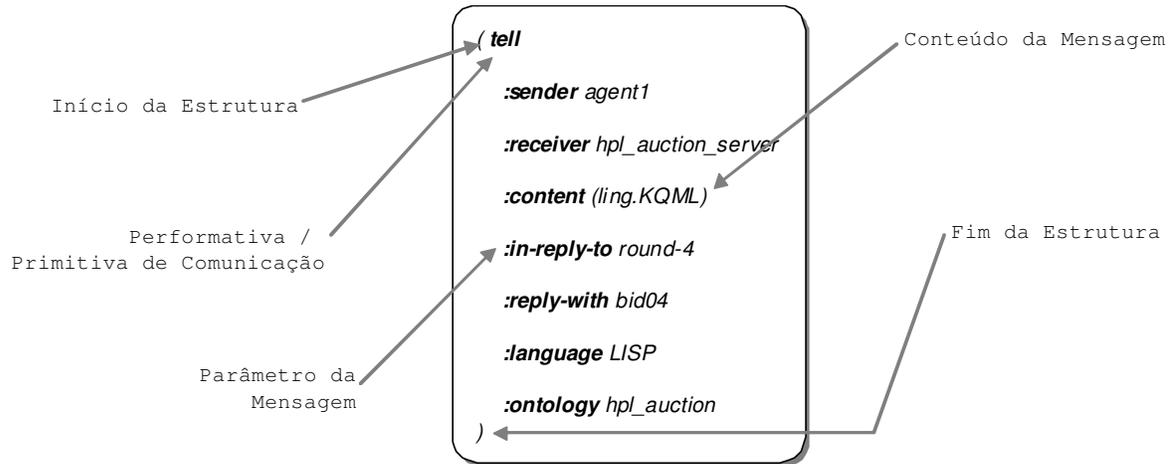


Figura 5.2: Componentes de uma mensagem KQML

5.3.4.2. ARQUITETURA KQML

Originalmente, KQML foi feita em duas versões interoperáveis, uma em C e outra em LISP. Esta implementação tem algumas características opcionais da linguagem: *programas especializados*, (um roteador e um facilitador) e uma *biblioteca de rotinas de interface* chamada KRIL (*KQML Router Interface Library*).

ROTEADORES KQML

Os Roteadores KQML são roteadores de mensagens independentes de conteúdo [11]. Cada agente que fala KQML está associado com seu próprio processo de roteamento.

Todos os roteadores são idênticos, pois são apenas cópias de um mesmo programa. O roteador manipula todas as mensagens enviadas e recebidas dos seus agentes associados, provendo assim uma maneira do agente comunicar-se com toda uma rede.

Os Roteadores KQML não manipulam o conteúdo das mensagens. Eles só se preocupam com as performativas e seus argumentos. Se uma mensagem especifica um endereço Internet, o roteador direciona a

mensagem para tal endereço. Se a mensagem especifica um serviço pelo nome, o roteador tentará achar o endereço Internet para aquele serviço e transmitirá a mensagem. Se uma mensagem somente provê a descrição do conteúdo, o roteador pode tentar achar um servidor que aceite a mensagem e então transmiti-la ou escolher transmitir para um agente de comunicação que esteja disposto a roteá-la (Figura 5.3).

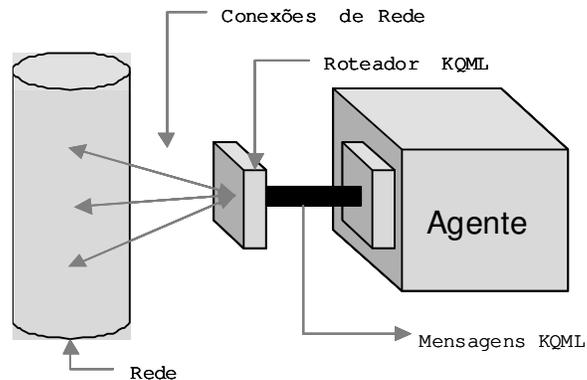


Figura 5.3: Modelo de um roteador para interface entre agentes e a rede

FACILITADORES OU MEDIADORES KQML

Um dos critérios de projeto do KQML era produzir uma linguagem que pudesse suportar uma grande variedade de arquiteturas de agentes. Para isso foi necessário introduzir uma classe especial de agentes chamada *facilitadores de comunicação*.

Um agente facilitador é uma aplicação que provê serviços de rede a outros agentes. Eles, além de manterem um registro de nomes de serviços, auxiliam os roteadores a encontrar *hosts* para rotear as informações. Neste caso, os facilitadores fazem o papel de consultores para o processo de comunicação.

No entanto, os facilitadores podem proporcionar muitos outros serviços de comunicação. Ao ser requisitado, o facilitador pode encaminhar mensagens para serviços cadastrados ou prover serviços de *match-making* entre os provedores de informação e os consumidores.

Para exemplificar algumas funcionalidades providas pelos facilitadores KQML, considere um caso onde o agente A gostaria de saber a autenticidade de uma sentença X, o agente B pode saber avaliar X, e um agente facilitador está disponível. Se A está ciente que B sabe avaliar X, então pode usar um protocolo simples ponto-a-ponto e enviar a questão *ask(X)* diretamente a B, como mostra a Figura 5.4. Porém, se A não está ciente quais agentes estão disponíveis, ou ainda, não sabe como contatar estes agentes, então uma variedade de primitivas podem ser usadas.

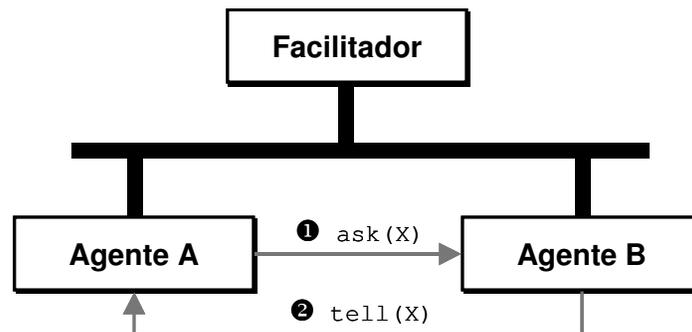


Figura 5.4: Comunicação usando protocolo simples ponto-a-ponto

A Figura 5.5 mostra um exemplo no qual A usa a primitiva *subscribe* para pedir que o facilitador monitore (avaliar) a autenticidade de X. Se posteriormente o Agente Facilitador recebe uma mensagem de B atestando a veracidade de X, A é notificado sobre X (*tell(x)*). Neste caso, para cada novo agente que informar (*tell(X)*) ao facilitador sobre a situação de X, o agente A ficará sabendo (pelo menos até o agente A fazer *unsubscribe(X)*). Nenhuma pergunta é feita a outros agentes que saibam avaliar X.

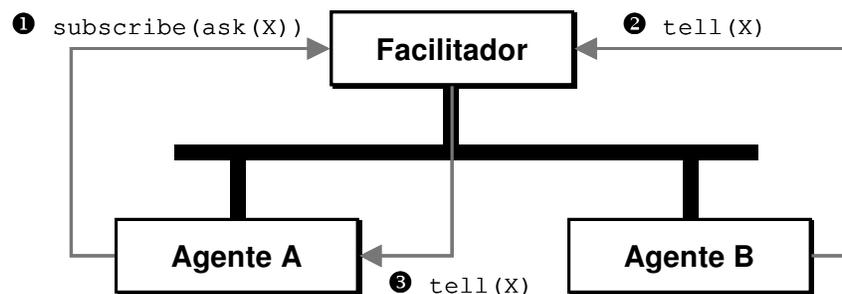


Figura 5.5: Uso da primitiva *subscribe*

A Figura 5.6 mostra uma situação ligeiramente diferente. O agente A pede para o agente facilitador que ache um agente, e somente um, que possa processar uma primitiva *ask(X)* (*broker-one(ask(X))*). Independente do pedido de A, o agente B informa ao facilitador que está disposto a aceitar primitivas que unificam com *ask(?)* (*advertise(ask(?))*), isto é, B está avisando que sabe processar mensagens do tipo *ask(?)*. Sabendo agora que B pode processar as requisições de A, o agente facilitador pergunta a B sobre X (*ask(X)*), recebe a resposta e a repassa para o agente A.

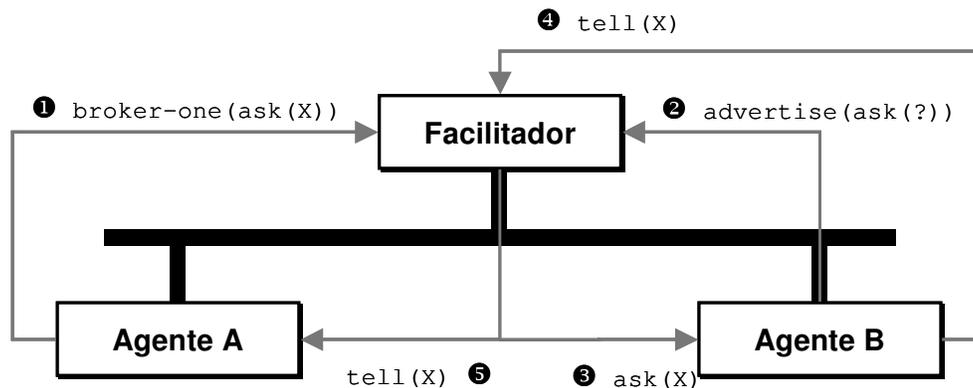


Figura 5.6: Uso da primitiva *broker-one*

Na Figura 5.7, o agente A usa uma primitiva ligeiramente diferente para informar ao facilitador de seu interesse em saber a verdade sobre X. A primitiva *recruit* pede para o receptor que ache um agente que esteja disposto a receber e processar uma primitiva embutida. A resposta daquele agente é então enviada diretamente ao agente iniciante, isto é, o facilitador envia a mensagem ao agente B (que sabe processar a mensagem) trocando a identidade do emissor para o agente A.

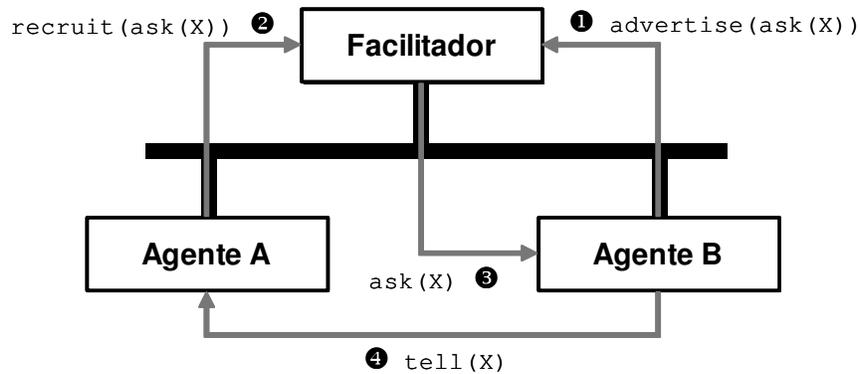


Figura 5.7: Uso da primitiva *recruit*

Destes exemplos, podemos concluir que uma das principais funções de agentes facilitadores é auxiliar outros agentes a encontrar os clientes e servidores apropriados. O problema de como os agentes encontram os facilitadores em primeiro lugar não é estritamente um assunto para KQML e tem uma variedade de possíveis soluções [60].

5.4. FIPA

A FIPA (*Foundation for Intelligent Physical Agents*) é uma fundação sem fins lucrativos cujo propósito é promover o sucesso de aplicações, serviços e equipamentos que utilizam a tecnologia de agentes [61]. Os objetivos da FIPA são fazer e disponibilizar especificações para maximizar a interoperabilidade entre sistemas de agentes.

FIPA ACL (*Agent Communication Language*) é a linguagem de comunicação de agentes da FIPA, desenvolvida como a linguagem de comunicação entre agentes de próxima geração, absorvendo as vantagens de KQML e solucionando as suas falhas. FIPA ACL é superficialmente similar a KQML. Suas sintaxes são idênticas, exceto por algumas primitivas de nomes diferentes. A linguagem externa (correspondente à camada de mensagem de KQML) define a intenção locutiva desejada da mensagem. A linguagem interna (correspondente à camada de conteúdo de KQML) denota as expressões para o conteúdo. As *primitivas de comunicação* ou *performativas* de KQML são tratadas como

atos comunicativos em FIPA ACL, apesar de se tratar do mesmo conceito.

No modelo semântico de FIPA ACL, os agentes não podem manipular diretamente a base de conhecimento virtual dos outros agentes, fazendo assim com que primitivas como *insert*, *uninsert*, *delete-one*, *delete-all*, *undelete* de KQML tornem-se sem sentido.

A arquitetura de FIPA tem um sistema chamado AMS (*Agent Management System*) que especifica serviços que controlam as comunidades de agentes, o mesmo papel dos roteadores e facilitadores de KQML. Isto elimina a necessidade de primitivas como *register/unregister*, *recommend*, *recruit*, *broker* e *(un)advertise* na LCA. Para ser mais preciso, estas primitivas foram movidas para o AMS.

Há ainda diferenças entre FIPA ACL e KQML que ainda não foram solucionadas. Não se sabe a dose certa do que deve ser feito pela linguagem externa e o que deve ser delegado à linguagem interna, por exemplo, um agente A pode falar para o agente B alcançar o objetivo X de duas maneiras diferentes:

1. Em KQML: *achieve goal X*

O agente A manda o agente B alcançar o objetivo X, escrevendo tudo no vocabulário da LCA.

2. Em FIPA ACL: *tell (achieve goal X)*

Aqui o agente A diz algo para o agente B que a LCA não entende. No entanto, o interpretador da linguagem interna reconhece o conteúdo da mensagem como um pedido para alcançar o objetivo X.

No caso de KQML (1), a linguagem externa tem uma performativa *achieve* específica para alcançar objetivos, já que a linguagem externa não assume que a linguagem de conteúdo tenha necessariamente esta capacidade. No caso de FIPA (2), a performativa *achieve* é "empurrada" para a linguagem interna, eliminando a necessidade da

mesma na linguagem externa. Na opinião de FIPA os objetos que fazem interações são partes da linguagem de conteúdo.

Apesar de parecer uma linguagem mais bem definida conceitualmente, FIPA ACL tem poucas implementações. KQML ainda é a linguagem *de facto* para comunicação entre agentes.

5.5. FERRAMENTAS

Depois de escolhida a linguagem de programação a ser utilizada neste trabalho (J2ME) e discutidos os principais requisitos para a implementação do componente de comunicação, escolhemos algumas ferramentas para avaliar. Esta avaliação tem por objetivo vislumbrar possíveis oportunidades de reaproveitamento. A Tabela 5.3 compara as características de uma série de ferramentas para comunicação de agentes.

FERRAMENTA	Agent Builder Lite	BOND	JACK	JACKAL	JADE	JAFMAS	JATLite	JKQML	MADKit	SACI	FIPA OS	ZEUS
Versão	1.3	2.1	2.1	3.1	2.01	0.1	0.4b	5.1a	2.0.1	RC3	1.3.2	1.05
Tipo	AC	B	AC	B	B	B	B	B	B	B	B	AC
Documentação	***	*	***	*	**	*	*	**	**	**	-	**
Protocolo de Transporte	RMI TCP/IP	Shadow	UDP/IP	TCP/IP	RMI	RMI	TCP/IP	KIP ATP OTP	??	RMI TCP/IP	RMI CORBA	TCP/IP
ACL	KQML	KQML	-	KQML	FIPA	KQML	KQML	KQML	KQML	KQML	FIPA	FIPA
Suporte a Ontologia	*	-	*	-	*	-	-	*	-	-	-	*
Suporte Roteador	-	-	*	-	-	-	*	*	-	*	-	*
Compatibilidade	*	**	**	**	**	**	**	**	**	**	**	*
Ambiente de Execução	*	-	*	-	-	-	-	-	*	-	-	*
Monitor/Depurador de Agentes	*	-	*	-	-	-	-	-	-	*	-	*
Controle de Projetos	*	-	*	-	-	-	-	-	-	-	-	*
Funcionalidades Extras	*	-	*	-	-	-	*	*	*	*	-	*
Licença	C, T	A, F, O	C, F, T	A, F, O	C, F, T	A, F, O	C, F, O	A, F, O				

Tabela 5.3: Tabela Comparativa entre as Ferramentas (AC=Ambiente Completo, B=Biblioteca, C=Comercial, A=Acadêmica, T=Avaliação ou para Fins Educativos, F=Gratuita, O=Código Aberto)

Depois de avaliadas as características das ferramentas, passamos para uma análise prática de algumas destas ferramentas e escolhemos o SACI por uma série de vantagens.

5.5.1. SACI

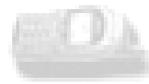
O SACI (*Simple Agent Communication Infrastructure*) é uma API Java para o auxílio no desenvolvimento de agentes capazes de se comunicar. O SACI é uma API Java de código aberto desenvolvida no Departamento de Engenharia de Computação e Sistemas Digitais da Universidade de São Paulo (USP). SACI define uma arquitetura baseada nos padrões de KQML e um modelo de sociedade. Os agentes das sociedades SACI podem ser referenciados pelo seu nome. O endereço dele na rede é completamente transparente para o programador. Há ainda a possibilidade de se implementar agentes como *applets* Java e rodá-los em *browsers*; executar remotamente os agentes das sociedades e monitorar eventos sociais (através de interfaces gráficas), tais como entrar e sair das sociedades, enviar e receber mensagens.

Além das boas características funcionais apresentadas por SACI, ainda pode-se observar vantagens no tocante a requisitos não funcionais, tais como performance [10], facilidade de uso, tamanho e facilidade de adaptação e/ou extensão. De fato, com exceção de RMI, o SACI não usa recursos mais elaborados da linguagem, possibilitando assim que seja facilmente adaptado e/ou estendido para a plataforma J2ME.

SACI foi a ferramenta escolhida para implementação da infraestrutura de comunicação que suporta agentes rodando em dispositivos móveis, pois além de todas as vantagens discutidas, tínhamos fácil acesso aos seus desenvolvedores, o que facilitaria uma possível adaptação do código original.

Como será discutido mais adiante, a arquitetura SACI foi estendida no sentido de suportar agentes escritos em J2ME, passando a se chamar por este motivo, KSACI. Para isso adaptações tiveram que ser feitas tais como a utilização de um servidor HTTP e de um *servlet* para processar as informações dos agentes implementados em

J2ME, uma vez que não há suporte a RMI, só HTTP. O KSACI tem ainda a possibilidade de se trocar estados de objetos Java através de conteúdos XML nas mensagens.



KSACI

Este capítulo descreve detalhadamente o trabalho de implementação do componente de comunicação para dispositivos móveis, o KSACI. O KSACI é a sigla em inglês de "*Simple Agent Communication Infrastructure*", onde o "K" significa algo desenvolvido para a máquina virtual J2ME, a KVM.

O KSACI é uma extensão da arquitetura do SACI, uma implementação de uma infra-estrutura de comunicação entre agentes baseada em KQML e escrita em J2SE. Com o KSACI os agentes podem ser implementados em J2ME e executar em dispositivos móveis, tais como celulares, *paggers* e *palm-tops*.

A seguir o SACI será detalhado com suas características, arquitetura e funcionalidades. Depois disto, o trabalho de extensão será explicitada com suas principais dificuldades, a arquitetura estendida, as funcionalidades extras e finalmente, um exemplo da utilização do KSACI.

6.1. SACI

Como já foi mencionado anteriormente, o SACI é uma ferramenta de código aberto com o propósito de prover a comunicação entre agentes usando KQML e em breve FIPA ACL. Basicamente o SACI proporciona aos agentes situados em um determinado ambiente, uma infra-estrutura

para interação e comunicação com os demais. Nesta infra-estrutura os agentes podem conhecer os outros agentes de uma sociedade consultando por sua identificação ou por suas habilidades. A seguir será explicado o modelo de sociedade utilizado pelo SACI.

6.1.1. MODELO DE SOCIEDADE SACI

As informações necessárias para o conhecimento mútuo entre os agentes formam a estrutura da sociedade. Formalmente, o estado da estrutura de uma sociedade SACI é definido pela tupla:

$$Soc = \langle A, S, l, \delta \rangle$$

onde

- $A = \{ \alpha \mid \alpha \text{ é a identificação de um agente que pertence à sociedade } \},$
- $S = \{ \sigma \mid \sigma \text{ é uma habilidade disponível na sociedade } \},$
- l é a linguagem utilizada na sociedade, e
- $\delta : A \rightarrow \wp(S)$ é uma função que mapeia as habilidades de um agente, tal que $\delta(\alpha) = \{ \sigma \mid \sigma \text{ é uma habilidade de } \alpha \}$

Por exemplo:

```
Society_A = <
    { Ed, Zé, Jô, Maria },
    { Java, C, Prolog, Pascal },
    { Português },
    { Zé → { Java, Prolog }, Jô → { Pascal }, Ed → { C }
>
```

Esta estrutura pode mudar no decorrer do tempo devido a eventos sociais que refletem o ciclo de vida de um agente. O ciclo de vida de um agente numa sociedade SACI tem dois estados: *ativo* e *inativo*.

Uma vez criado, o agente fica no estado "inativo", pois ainda não está presente em nenhuma sociedade. Ao entrar em uma sociedade, o agente passa para o estado "ativo", podendo assim enviar e receber mensagens e anunciar suas habilidades. Ao finalizar suas atividades e sair da sociedade, o agente passa novamente para o estado "inativo", não podendo mais executar nenhuma tarefa até que esteja novamente em uma sociedade.

Para *entrar* em uma sociedade, o agente deve adicionar sua identificação ao banco de nomes de agentes da sociedade. A entrada de um agente altera a estrutura da sociedade da seguinte forma:

$$\langle A, S, I, \delta \rangle_i \Rightarrow \langle A', S, I, \delta \rangle_{i+1} \mid A' = A \cup \{ \alpha \}$$

O *envio e recebimento* de mensagens entre agentes da mesma sociedade não alteram a estrutura da sociedade. Mas para *anunciar as habilidades* os agentes devem cadastrá-las na base de habilidades da sociedade. O cadastro de habilidades de um agente α muda a estrutura da sociedade da seguinte forma:

$$\langle A, S, I, \delta \rangle_i \Rightarrow \langle A, S', I, \delta' \rangle_{i+1}$$

Onde

$$S' = S \cup \{ \sigma \}$$

$$\delta'(x) = \begin{cases} \delta(x) & \text{se } x \neq \alpha \\ \delta(\alpha) \cup \{ \sigma \} & \text{se } x = \alpha \end{cases}$$

Para *deixar* a sociedade, os agentes devem abdicar de suas funções dentro da sociedade. A saída de um agente α da sociedade causa uma mudança na estrutura da mesma da seguinte forma:

$$\langle A, S, I, \delta \rangle_i \Rightarrow \langle A', S', I, \delta' \rangle_{i+1}$$

Onde

$$A' = A - \{ \alpha \}$$

$$S' = \{ \sigma \mid \sigma \in \delta'(x) \}$$

$$\delta'(x) = \begin{cases} \delta(x) & \text{se } x \in A' \\ \{ \} & \text{se } x \notin A' \end{cases}$$

6.1.2. ARQUITETURA SACI

A arquitetura do SACI é baseada no paradigma cliente servidor, onde cada servidor pode manipular várias sociedades. Como sugerido pela arquitetura de KQML, cada sociedade SACI tem um, e somente um, facilitador (Fac.Soc1 na Figura 6.1) visto na forma de um agente especial possuindo na sua estrutura interna, uma identificação, uma localização e os serviços disponibilizados aos agentes da sociedade.

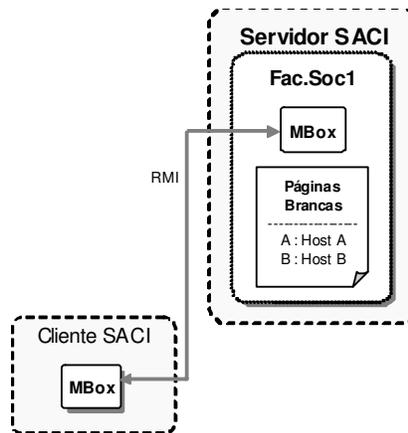


Figura 6.1: Arquitetura SACI

Os agentes facilitadores são os responsáveis, como o próprio nome sugere, por facilitar o processo de comunicação entre os agentes. O facilitador tem papel fundamental no modelo de sociedade proposto pelo SACI, isto é, na entrada e saída dos agentes nas sociedades, nos anúncios de habilidades e principalmente nas trocas de mensagens.

6.1.2.1. ENTRAR E DEIXAR A SOCIEDADE

Para entrar numa sociedade o agente deve contactar o facilitador da sociedade e pedir o seu registro. Para registrar os agentes, o facilitador associa o nome do agente à sua localização numa lista nomeada *páginas brancas*. As páginas brancas funcionam basicamente como suas homônimas nas listas telefônicas, onde os assinantes são associados aos seus telefones. Após este registro, o facilitador gera uma identificação única para este agente dentro da sociedade.

Para sair da sociedade, o agente deve requisitar ao facilitador que remova suas referências das páginas brancas e amarelas que serão detalhadas mais adiante.

6.1.2.2. ENVIAR E RECEBER MENSAGENS

Para enviar mensagens para outros agentes da sociedade, os agentes possuem um componente chamado *Mbox* (de *message box*, ou caixa de mensagens) que serve de interface entre eles e a sociedade. Sua finalidade é tornar transparente o envio e o recebimento de mensagens. Este componente possui funções que encapsulam a composição de mensagens KQML, o envio síncrono e assíncrono de mensagens, o recebimento de mensagens, o anúncio e consulta de habilidades e o *broadcast* de mensagens. Para estar presente em mais de uma sociedade, o agente deve possuir um *Mbox* para cada uma dessas sociedades, ou seja, cada sociedade tem no máximo uma *Mbox* para cada agente.

Para exemplificar o processo de troca de mensagens num servidor de sociedades SACI, considere a Figura 6.2 onde existem 3 *hosts* diferentes: o *host* C abriga o servidor da sociedade *Soc1* (e conseqüentemente o seu agente facilitador *Fac.Soc1*) e os *hosts* A e B abrigam respectivamente os agentes A e B. Na situação proposta na figura, após o registro independente dos dois agentes (1) e (2) na sociedade *Soc1*, o agente A gostaria de enviar uma mensagem ao agente B, mas não conhece a sua localização exata. O agente A deve então fazer uma consulta a *Fac.Soc1*, cujo endereço já deve ser conhecido (3). O facilitador então consulta suas páginas brancas e responde a A o endereço de B (4). Conhecendo agora a localização de B, A pode enviar mensagens diretamente a B (5). Em outra situação, se A não quisesse ou não pudesse saber o endereço de B ele poderia delegar ao facilitador o envio das mensagens.

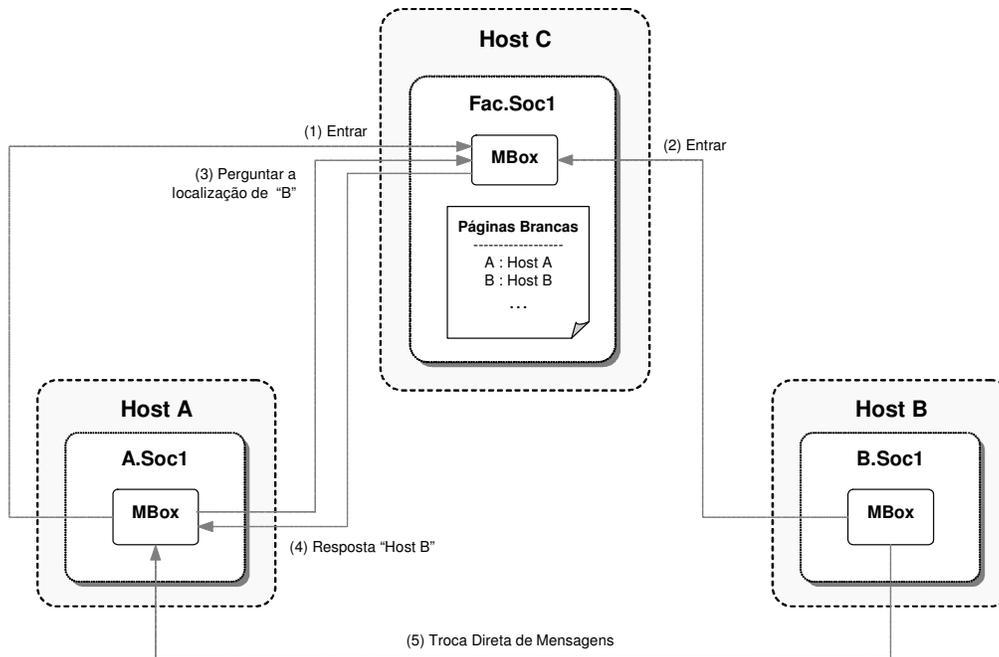


Figura 6.2: Troca de mensagens utilizando o facilitador

6.1.2.3. CADASTRANDO HABILIDADES

Como mencionado no modelo de sociedade SACI, os agentes podem anunciar habilidades e para isso utilizam uma funcionalidade do facilitador que se chama *páginas amarelas*, que funciona também como suas homônimas nas listas telefônicas, onde os assinantes são listados não pelos seus nomes, mas por suas habilidades. Nas páginas amarelas, o facilitador faz uma associação dos agentes com suas habilidades dentro da sociedade.

Para exemplificar o anúncio de habilidades numa sociedade SACI, considere a Figura 6.3 e suponha que os agentes A e B já foram devidamente registrados na sociedade *Soc1* do *Host C*. Quando um agente não está interessado em saber o nome e nem a localização de um agente específico e sim em quem pode processar determinada requisição, ele deve enviar uma mensagem KQML ao facilitador (2) similar a apresentada na Listagem 6.1 abaixo.

```
( recommend-all
  :receiver Facilitator
```

```

:sender      A
:reply-with idl
:language   KQML
:ontology   yellow-pages
:content    ( ask-one
                :language algebra
                :ontology math
                :content  "X + Y"
              )
)

```

Listagem 6.1: Mensagem do agente A para o Facilitador

Que significa que o agente A quer saber quais agentes podem processar uma mensagem similar a mensagem KQML descrita no conteúdo (*content*). Independentemente, o facilitador recebe de B a mensagem pedindo o cadastro das suas habilidades nas páginas amarelas (ver Listagem 6.2).

```

( advertise
  :receiver   Facilitator
  :sender     B
  :language   KQML
  :ontology   yellow-pages
  :content    ( ask-one
                  :receiver B
                  :language algebra
                  :ontology math
                  :content  "X + Y"
                )
)

```

Listagem 6.2: Mensagem do agente B para o Facilitador

Sabendo agora que o agente B pode processar mensagens do tipo requerido por A, o facilitador pode responder ao agente A com a seguinte mensagem (3) (ver Listagem 6.3).

```

( tell
  :receiver      A
  :sender        Facilitator
  :in-reply-to  id1
  :language      KQML
  :ontology      yellow-pages
  :content       (B)
)

```

Listagem 6.3: Mensagem do Facilitador para o agente A

De posse do endereço do agente que pode processar mensagens pedindo para somar dois números ($X+Y$), o agente A pode enviar mensagens diretamente a B (4).

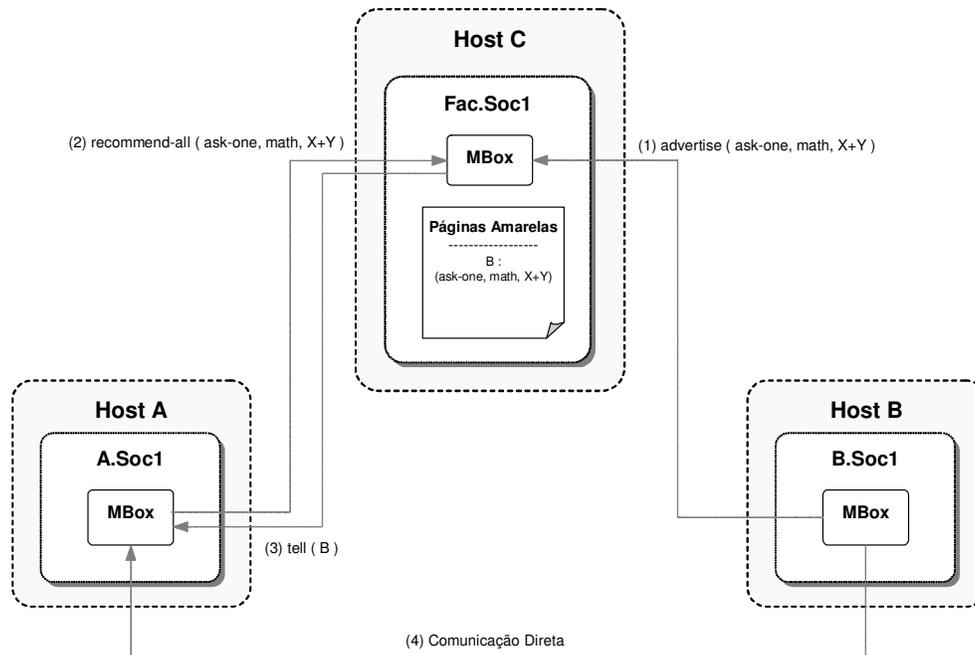


Figura 6.3: Anúncio de habilidades

6.2. PROCESSO DE EXTENSÃO

Depois de explicados o modelo de sociedade e a arquitetura do SACI, esta seção discutirá o processo de extensão dessa arquitetura

para a produção do KSACI, sendo, portanto uma das contribuições deste trabalho.

6.2.1. DIFICULDADES

Numa sociedade SACI, todo o tipo de comunicação (agente-agente e agente-servidor) é feita através do protocolo de transporte RMI [51]. A implementação de RMI utiliza o conceito de serialização de objetos Java, que por sua vez faz uso da reflexão. Como visto no capítulo 4, J2ME não suporta reflexão e por conseqüência nenhum dos recursos derivados dela. Esta limitação da API de J2ME produz dois grandes obstáculos para o processo de extensão:

- A impossibilidade do uso das facilidades de RMI;
- A não serialização de objetos Java;

As próximas duas seções esclarecerão como este trabalho lidou com estas dificuldades durante o processo de extensão da arquitetura SACI.

6.2.2. DE RMI PARA HTTP

A impossibilidade do uso de RMI fez com que a arquitetura do KSACI tivesse que usar o único protocolo de transporte disponível na API de J2ME: o HTTP. Para isso, faz-se necessário o uso de um servidor HTTP para "escutar" as requisições. Além desse servidor HTTP seria necessário uma aplicação rodando junto com o servidor para processar as requisições. Várias são as possibilidades para se implementar esta aplicação, dentre elas: *CGI* [62], *Perl* [63], *PHP* [64], *ASP* [65], *Servlet* [66], *JSP* [67]. Uma vez que o SACI é implementado em Java, a opção natural seria a utilização de *Servlet Java*.

Esta aplicação será responsável por criar, para cada agente inserido na sociedade, um agente especial chamado *Agente Proxy* (representação do agente requisitante) e inseri-lo na devida sociedade SACI (ver Figura 6.4). Ela também deverá ser capaz de

redirecionar as outras requisições às suas respectivas funcionalidades na sociedade SACI. A comunicação deste *Servlet* e o servidor, assim como a dos *Agentes SACI*, será feita usando RMI (ver Figura 6.4).

Para os agentes rodando em dispositivos móveis (*Agente KSACI*), foi definida uma API semelhante a do SACI, no entanto, mais reduzida.

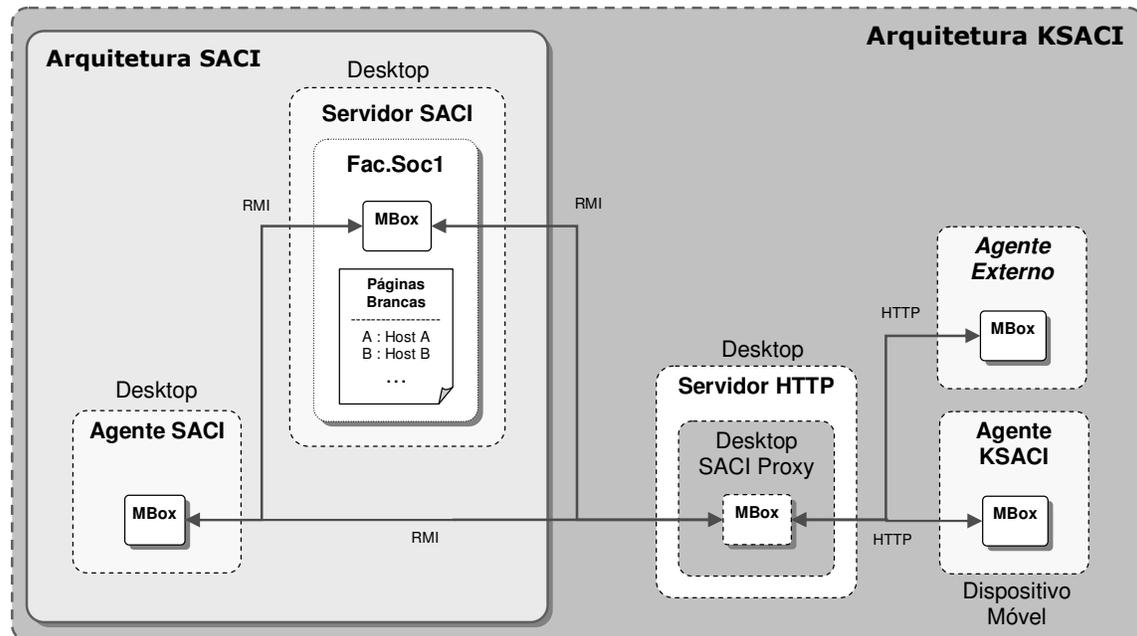


Figura 6.4: Arquitetura KSACI

O Agente KSACI é representado pela classe *kAgent* retratada na Figura 6.5. Ela implementa a interface *Runnable* de J2ME que permite que os agentes possam ser executados por uma *Thread*. Assim sendo, os agentes KSACI têm a possibilidade de executar em *background* um trecho de código enquanto eles estejam ativos através da implementação do método *run()*.

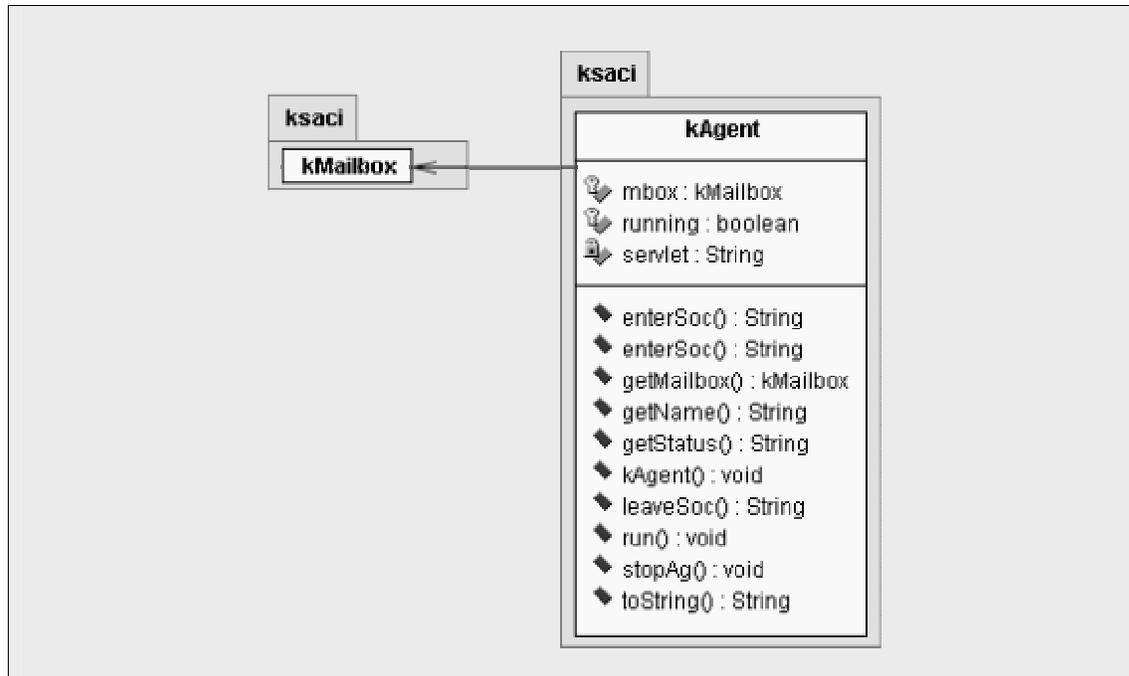


Figura 6.5: kAgent Diagram

Cada Agente KSACI tem uma caixa de mensagens equivalente a *MBox* do SACI representada pela classe *kMailbox* na Figura 6.6. A classe *kMailbox* é a interface entre o agente KSACI e o Servidor HTTP, e conseqüentemente a Sociedade SACI. Esta classe é quem controla a entrada (*enterSoc*) e saída (*leaveSoc*) dos agentes nas sociedades e troca de mensagens KQML entre os agentes e o servidor (*sendMsg*, *readMsg*, *readXMLMsg* e *sendXMLMsg*). As informações de identificação dos agentes, tais como, *nome*, *sociedade*, *identificador* e o *endereço do servidor HTTP*, estão guardadas nos seus respectivos *kMailbox*.

A comunicação com o servidor HTTP é feita através de um protocolo muito simples: uma URL é montada com informações relevantes ao comando que se quer efetuar e enviada ao servidor. O *servlet* vai então receber a requisição, avaliar os atributos e seus valores, e redirecionar a interpretação para um método específico. Um desses atributos avaliados é o "*command*" que indica qual a operação está sendo feita (e.g. *command=enterSoc*, *command=sendMsg*). Os outros atributos funcionam como argumentos para o comando especificado (e.g. *name=agentel*, *soc=socl*, *id=agentel@socl*).

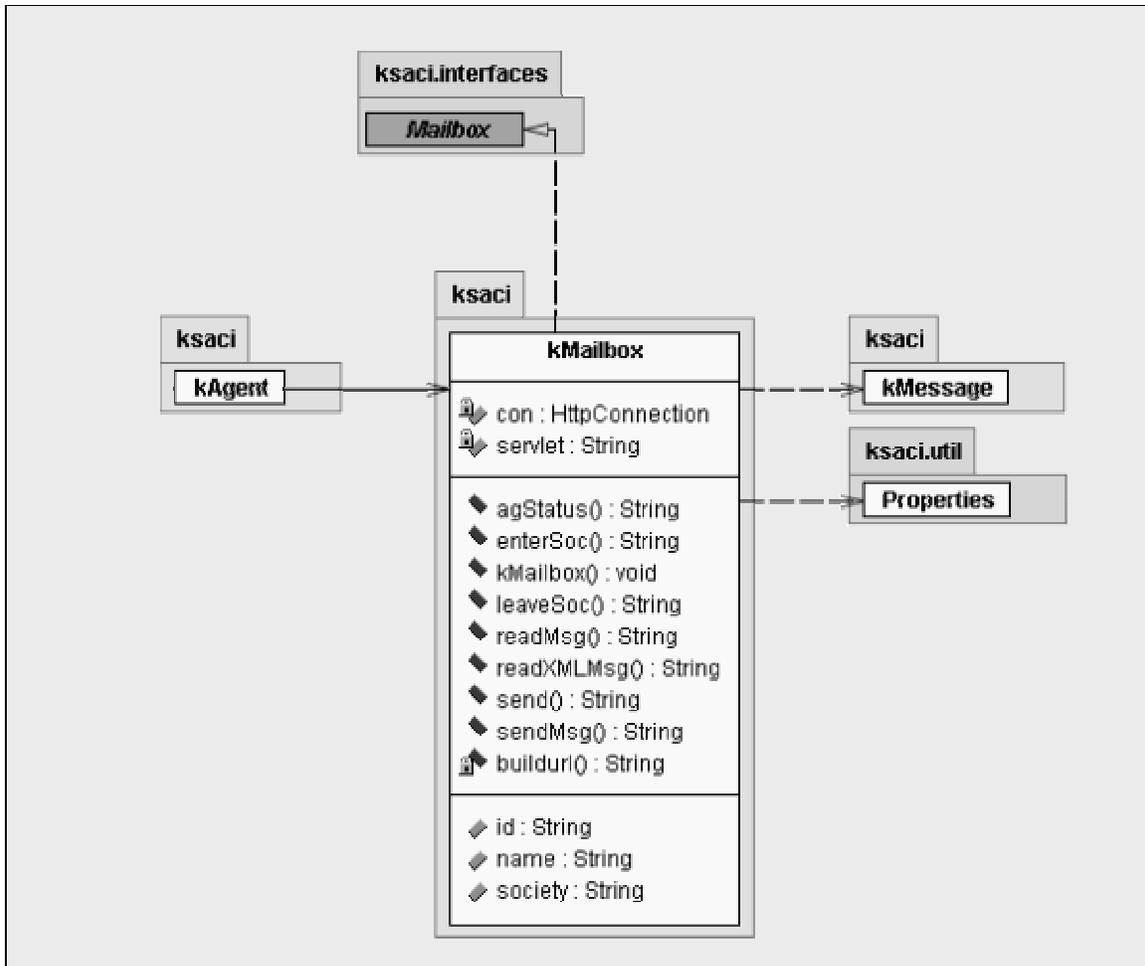


Figura 6.6: kMailbox Diagram

No momento da requisição de entrada na sociedade, o servlet armazena um *Agente Proxy*, que representa o agente requisitante, numa lista estática de *Agentes SACI*. Todas as demais requisições feitas por este agente requisitante serão redirecionadas a este *Agente Proxy* identificado pelo atributo "id" das requisições. É importante notar que a sintaxe de todos os atributos das requisições deve ser a mesma adotada tanto pelo *Servlet* quanto pelo agente.

Para a troca de mensagens KQML é utilizada uma classe *kMessage* que representa uma mensagem KQML. Ela foi implementada como uma extensão da classe *Hashtable* de Java de forma que as chaves representam os atributos KQML e os elementos, o valor dos atributos (Figura 6.7).

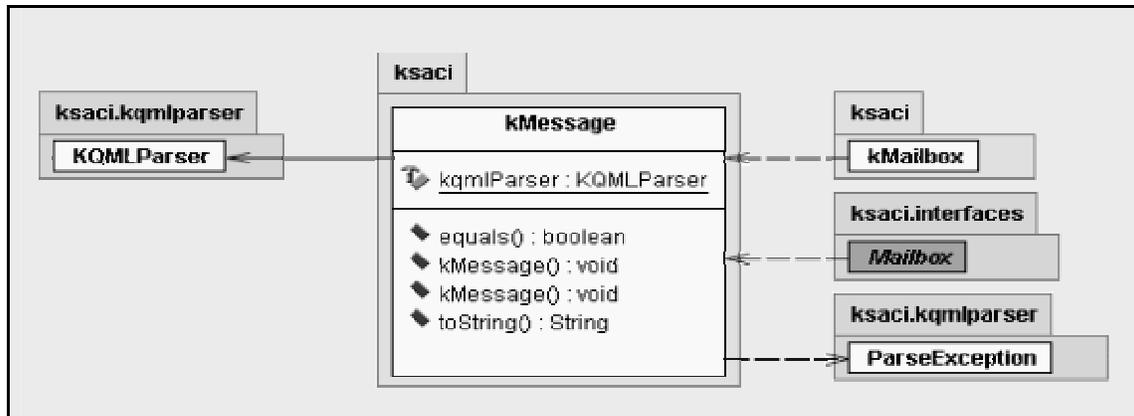


Figura 6.7: kMessage Diagram

Os trechos de código da Listagem 6.5, Listagem 6.6 e Listagem 6.7 são opções para a criação de um objeto da classe *kMessage* que representa a mensagem QXML da Listagem 6.4.

```
( ask-one
  : content "2+4"
  : receiver APlusServer
  : reply-with rAdd
)
```

Listagem 6.4: Mensagem QXML

```
kMessage m = new kMessage ( "
  ( ask-one :content \"2+4\"
    :receiver APlusServer
    :reply-with rAdd
  )
  " );
```

Listagem 6.5: Criação de um objeto *kMessage* (Opção #1)

```
kMessage m = new kMessage ();
m.put ( "performative", "ask-one" );
m.put ( "content", "\"2+4\"" );
m.put ( "receiver", "APlusServer" );
m.put ( "reply-with", "rAdd" );
```

Listagem 6.6: Criação de um objeto *kMessage* (Opção #2)

```
kMessage m = new kMessage ("
    ( ask-one :content \"2+4\" )
");
m.put ( "receiver", "APlusServer" );
m.put ( "reply-with", "rAdd" );
```

Listagem 6.7: Criação de um objeto *kMessage* (Opção #3)

No caso das Listagem 6.5 e Listagem 6.7, onde o construtor de *kMessage* recebe uma *String* como parâmetro, haverá a necessidade de um *parser* KQML para a análise e validação desta *String*. Se toda a sintaxe estiver coerente, um novo objeto *kMessage* que representa a mensagem KQML acima será criado.

6.2.3. SERIALIZAÇÃO

A *serialização* é o processo pelo qual os objetos Java são decompostos e escritos em algum lugar, ou seja, os estados dos objetos Java são escritos em um *socket* de rede ou em um arquivo, de forma que possam ser *de-serializados* em outro lugar. A serialização de objetos não poderá também ser usada na implementação do KSACI devido à ausência de reflexão em J2ME, mais especificamente no par CLDC 1.0 / MIDP 1.0.

É importante notar que somente o estado dos objetos (atributos não estáticos) são serializados. O seu comportamento (métodos) não são serializados. Com isso, faz-se necessário que a classe do objeto a ser serializado (e de-serializado) esteja presente no caminho das classes conhecidas da aplicação (*classpath*).

Analisando mais a fundo, pode-se visualizar uma estrutura onde o estado dos objetos pode estar dependente de outros objetos, ou seja, alguns destes estados são também objetos com estado próprio. O mecanismo de serialização Java trata estes casos serializando recursivamente todos os objetos dependentes. Na situação da Figura 6.8, existe uma classe com quatro atributos, sendo que dois deles (*objeto1* e *objeto2*) são também objetos. O mecanismo de serialização

irá serializar o estado do objeto representado pela classe *Classe1* e recursivamente os estados de *objeto1* e *objeto2*.

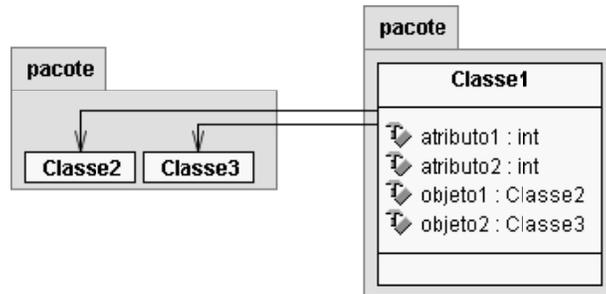


Figura 6.8: Rede de dependências entre objetos Java

Infelizmente, o mecanismo da serialização Java não está disponível em J2ME. O seu uso seria interessante na transmissão de mensagens cujo conteúdo é um objeto (ou o estado dele).

Para amenizar este problema, a arquitetura do KSACI provê um mecanismo baseado em XML para transmissão de estados simples de objetos. As classes que pretendem ter seu estado serializado devem estar no *classpath* do servidor SACI e no dispositivo onde a aplicação está rodando. Além disso, devem implementar a interface *ksaci.interfaces.MappedClass* (Figura 6.9) provendo código para os métodos *marshal()* e *unmarshal()*.

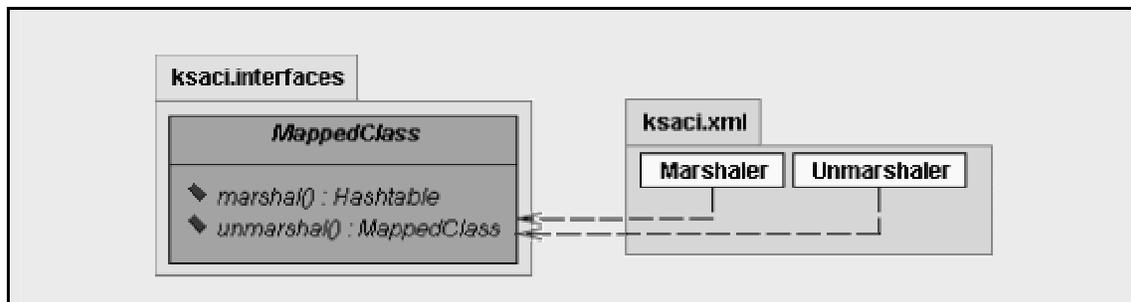


Figura 6.9: Diagrama da classe MappedClass

Para o método *marshal()*, o desenvolvedor deve retornar uma *Hashtable* onde as chaves são os nomes dos atributos e os elementos são os valores ou referências dos atributos. Para o método *unmarshal()* o desenvolvedor deve ler uma *Hashtable* com o mesmo conteúdo da *Hashtable* retornada pelo método *marshal()* e produzir um objeto da classe *MappedClass*.

Para fazer realmente as devidas transformações, são utilizados dois objetos: *Unmarshaler* e *Marshaler* ambos do pacote *ksaci.xml*. A classe *Unmarshaler* (Figura 6.10) tem um único método estático chamado *unmarshal()*, que recebe uma *String* (o documento XML equivalente) e retorna um objeto da classe *MappedClass*. A classe *Marshaler* (Figura 6.11) também possui um único método estático chamado *marshal()* que recebe um objeto da classe *MappedClass* e retorna o documento XML equivalente.

A arquitetura do KSACI possui um esquema extensível para suporte de tipos de atributos. Os desenvolvedores podem implementar seus próprios suportes a atributos complexos, i.e., para cada tipo de atributo, existirá um correspondente *marshaler* e *unmarshaler* que são usados internamente para fazer as conversões específicas (ver Figura 6.10 e Figura 6.11). Por exemplo, para a classe *String* têm-se *StringMarshaler* (que retorna a representação XML de uma dada *String*) e *StringUnmarshaler* (que retorna uma *String* codificada em um trecho de código XML).

As conversões *XML-Objeto* e *Objeto-XML* podem ser feitas das seguintes formas:

- XML-Objeto:

```
MappedClass m = Unmarshaler.unmarshal ( XMLDoc );
```

- Objeto-XML:

```
String xml = Marshaler.marshal ( MappedClass );
```

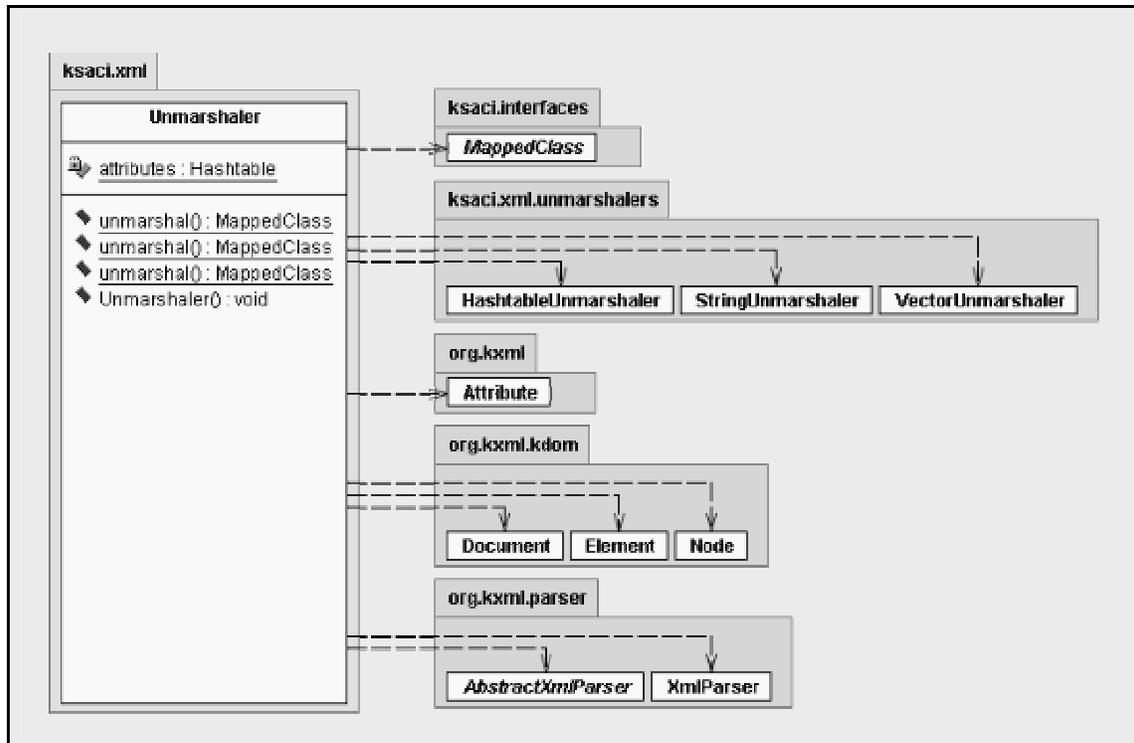


Figura 6.10: Diagrama da classe Unmarshaller

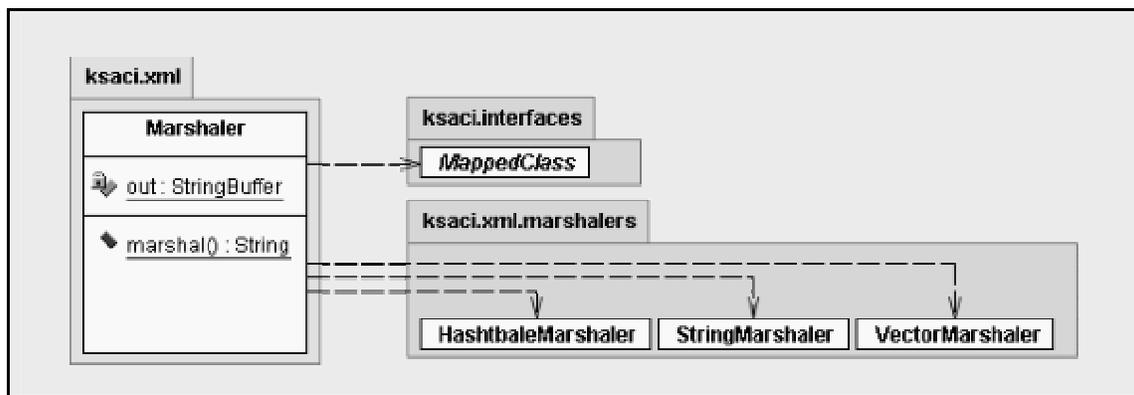


Figura 6.11: Diagrama da classe Marshaler

6.3. FUNCIONAMENTO

O funcionamento de uma aplicação KSACI é baseado na construção de um objeto da classe *kAgent* ou subclasse dela. Este objeto é a representação do agente que está rodando no dispositivo. Para efetivamente realizar alguma ação, o agente deve estar inserido em uma sociedade e para isso o objeto criado deve solicitar ao *Servlet*

localizado no servidor HTTP sua entrada. Após receber uma identificação única, sinal da sua aprovação na sociedade, o agente estará apto a trocar mensagens e anunciar habilidades.

Nesta seção serão detalhados dois exemplos da utilização do KSACI. O primeiro exemplo mostra uma aplicação onde um agente localizado no servidor tem a habilidade de somar dois números e outro agente localizado no dispositivo, faz solicitações a este servidor. O segundo exemplo mostra a utilização da troca de estados simples de objetos utilizando XML.

6.3.1. EXEMPLO 1

Suponha a classe *Client* estendendo a classe *kAgent* e implementando o método *run()*. Esta classe *Client* é cliente de um servidor que soma dois números. A implementação do método *run()* monta uma mensagem KQML e a envia ao servidor. O servidor responde e o método *run()* armazena a resposta na variável *resp* da soma:

```
public void run(String expression) {
    this.exp = expression;
    new Thread(this).start();
}

public void run() {
    try {
        // Construindo uma mensagem KQML
        kMessage m = new kMessage( "(ask-one)" );
        m.put("performative", "ask-one");
        m.put("receiver", "server");
        m.put("language", "algebra");
        m.put("ontology", "matemática");
        m.put("content", this.exp);

        // Enviando a mensagem KQML
        this.getMailbox().sendMsg(m, false);

        // Esperando a resposta do servidor
        String resp;
```

```

do {
    resp = this.getMailbox().readMsg();
} while (resp.indexOf( "no" ) != -1);

// Montando a resposta
kMessage answer = new kMessage( resp );
if (answer != null) {
    response = (String) answer.get("content");
}
} catch (Exception e) { }
}

```

Listagem 6.8: Parte da Implementação de uma agente KSACI

Para executar a classe *Client* deve-se usar um *MIDlet*. Para isso a classe *ClientMIDlet* (abaixo) utiliza um agente *Client* para fazer a requisição de soma de dois números. A requisição é feita e a resposta impressa no visor do aparelho.

```

public class ClientMIDlet extends MIDlet {
    private Client agent = new Client();
    public PlusMIDlet() {
        String res = agent.enterSoc( "client", "soc" );
        if (res.indexOf( "ok" ) != -1 ) {
            agent.run( "\\1+2\\" );
            String res = agent.getResponse();
            Alert a = new Alert(
                "Answer",
                res,
                null,
                AlertType.INFO
            );
            Display.getDisplay ( this ).setCurrent(a);
        }
    }
}

```

Listagem 6.9: MIDlet para execução de um agente KSACI

As figuras a seguir mostram a execução do exemplo anterior utilizando-se o emulador da Sun [49].

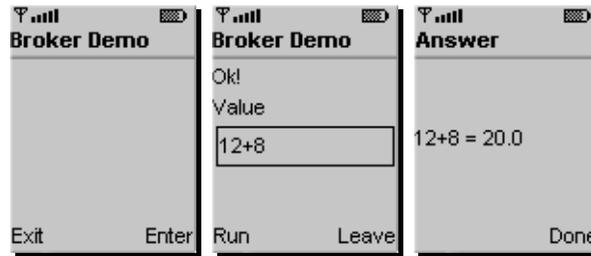


Figura 6.12: Telas do MIDlet que soma dois números

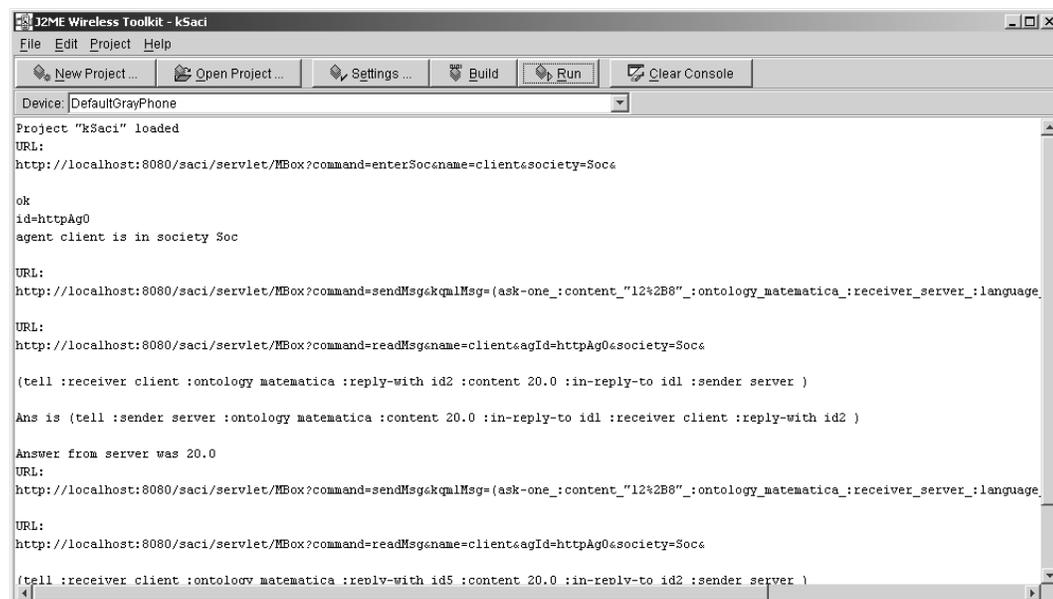


Figura 6.13: Output da execução no Sun J2ME Wireless Toolkit

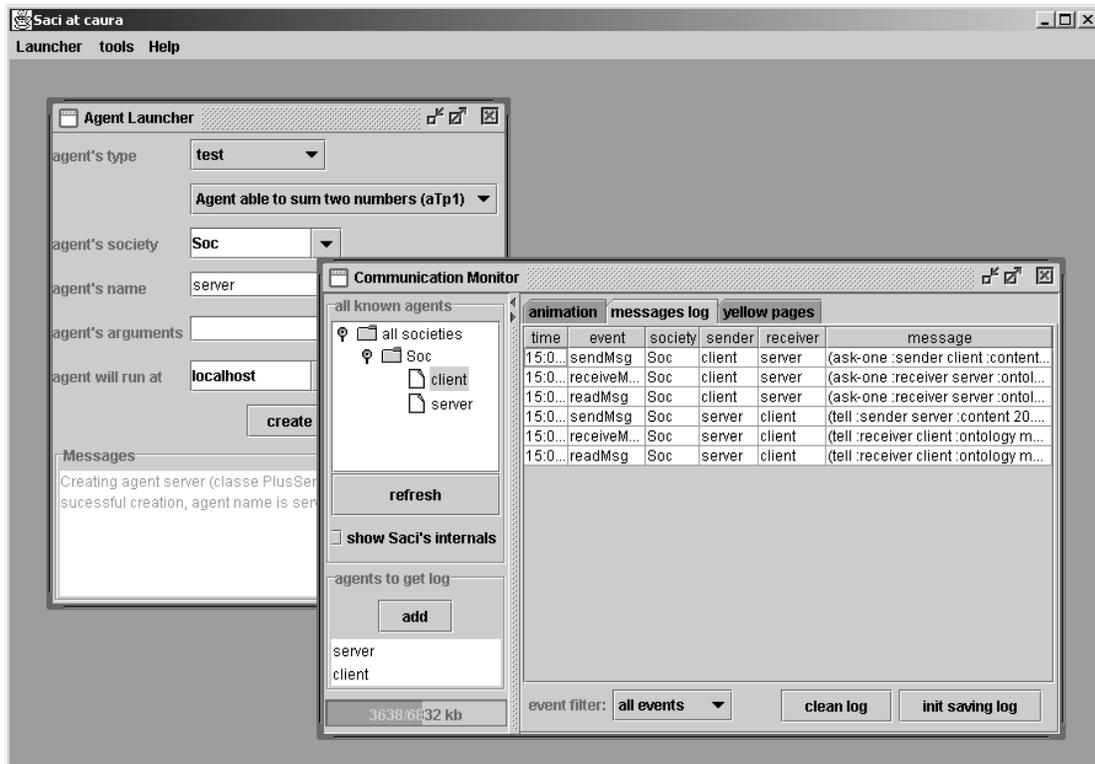


Figura 6.14: Ambiente de monitoração do SACI

6.3.2. EXEMPLO 2

Para transferir o estado dos objetos em mensagens KQML, o KSACI codifica este estado em XML. Esta codificação só pode ser realizada se o objeto cujo estado a ser transmitido, implementar a interface *MappedClass*, como descrito em 6.2.3. Para exemplificar a transformação de estados de objetos em XML e vice-versa, considere a classe *Pessoa* com os atributos *nome*, *idade*, *sexo* e *fonos* ilustrada na Listagem 6.10.

```

public class Pessoa {
    private String sexo = null;
    private String idade = null;
    private String nome = null;
    private Vector fonos = new Vector();
}

```

Listagem 6.10: Classe *Pessoa*

Para que esta classe possa ter seu estado transmitido, precisa-se implementar a interface *MappedClass* provendo código para os métodos *marshal()* e *unmarshal()*. A Listagem 6.11 ilustra a implementação destes dois métodos para a classe *Pessoa*, considerando que as devidas classes auxiliares foram importadas corretamente e que os métodos de acesso aos atributos foram corretamente definidos.

```
public Hashtable marshal() {
    Hashtable out = new Hashtable();
    out.put( "sexo", this.getSexo() );
    out.put( "nome", this.getNome() );
    out.put( "idade", this.getIdade() );
    out.put( "fones", this.getFones() );
    return out;
}

public MappedClass unmarshal( Hashtable xml ) {
    Enumeration key = xml.keys();
    Enumeration val = xml.elements();
    while ( key.hasMoreElements() ) {
        String _key = (String) key.nextElement();
        Object _val = val.nextElement();
        if ( _key.equals( "nome" ) ) {
            this.setNome( (String) _val );
        } else if ( _key.equals( "sexo" ) ) {
            this.setSexo( (String) _val );
        } else if ( _key.equals( "idade" ) ) {
            this.setIdade( (String) _val );
        } else if ( _key.equals( "fones" ) ) {
            this.addFones( (Vector) _val );
        }
    }
    return this;
}
```

Listagem 6.11: Classe Pessoa

Observe que o método *marshal()* retorna uma *Hashtable* que representa o estado do objeto e será utilizado pela classe *Marshaler* para proceder a transformação real para XML, como descrito abaixo:

```
String xml = Marshaler.marshal ( pessoa );
```

Da forma análoga, o método *unmarshal()* recebe um estado de objeto codificado em uma *Hashtable* e altera o estado do objeto. Este método é utilizado pela classe *Unmarshaler* para proceder a instanciação real do objeto, como descrito abaixo

```
Pessoa pessoa = (Pessoa) Unmarshaler.unmarshal ( XMLDoc );
```

Para utilizar o KSACI para trocar estes estados de objeto da classe *Pessoa*, o desenvolvedor deverá disponibilizá-la em todos os ambientes a serem utilizados, ou seja, nos *classpaths* dos clientes. A mensagem KQML deve informar ao servidor qual a classes sendo transferida com o atributo `":class"` seguido do nome da classe completo.



7.

RACIOCÍNIO

Neste capítulo serão tratados assuntos relacionados ao desenvolvimento do componente de raciocínio para dispositivos móveis. Serão discutidas as diversas maneiras de se embutir um raciocínio dedutivo em agentes rodando em dispositivos móveis, assim como os impactos da utilização de J2ME como linguagem de programação. Por fim serão analisadas duas ferramentas para possíveis reutilizações na implementação do KEOPS.

De acordo com a nossa experiência no desenvolvimento de aplicações embutidas utilizando J2ME [08] e de sistemas de produção orientados a objetos [08], identificamos três possíveis abordagens para a implementação desse tipo de raciocínio em plataformas móveis (dispositivos móveis) que serão descritas e discutidas na próxima seção.

7.1. DEDUÇÃO EM DISPOSITIVOS MÓVEIS

As três possibilidades identificadas por nós, foram as seguintes: a utilização de uma linguagem de programação orientada a IA (LPOIA) embutida no sistema operacional hospedeiro, a utilização de interfaces para a interoperabilidade entre sistemas e a utilização de uma linguagem de programação hospedeira provida por

uma máquina virtual para construção dos mecanismos de raciocínio. A seguir cada uma das possibilidades serão analisadas.

7.1.1. LPOIA EMBUTIDA NO SISTEMA OPERACIONAL HOSPEDEIRO

A primeira abordagem para se embutir o raciocínio dedutivo em dispositivos móveis é embutir uma linguagem de programação orientada a IA (LPOIA), que provê todas as funcionalidades necessárias para o raciocínio dedutivo (ver Figura 7.1). Esta linguagem pode ser de um dos seguintes tipos:

- Uma linguagem de programação do paradigma lógico, como por exemplo, *Prolog* [23].
- Uma linguagem de programação orientada a agentes, tais como *Agent0* [71], *METATEM* [72] e *Placa* [73]
- Uma linguagem de programação multi-paradigma, tais como *ObjLog* [74], *LIFE* [75], *JACK* [76] e *KIEV* [77]

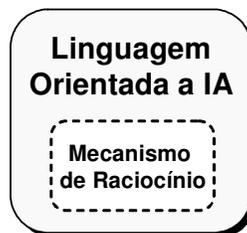


Figura 7.1: Linguagem orientada a IA embutida no sistema operacional hospedeiro

Esta abordagem demanda a construção ou adaptação de um compilador para gerar código de máquina do dispositivo a partir de um código-fonte. Este compilador deve transformar o código-fonte de uma LPOIA em *bytecodes* para serem interpretados por um interpretador que roda em cima do sistema operacional hospedeiro (para linguagens interpretadas) ou em código de máquina para executar em cima do próprio sistema operacional.

A construção de tais interpretadores ou compiladores é inviável uma vez que ela demandam novas implementações para cada dispositivo.

Mesmo que estas implementações não fossem o problema, elas teriam ainda de ser disponibilizadas junto com o sistema operacional dos dispositivos, o que demanda complicadas negociações industriais devido a problemas operacionais e de propriedade intelectual.

7.1.2. INTERFACES PARA INTEROPERABILIDADE

A segunda abordagem é baseada no desenvolvimento de interfaces para a interoperabilização dos agentes embutidos (implementados numa linguagem hospedeira) e um servidor capaz de raciocinar (implementado em uma linguagem de programação orientada a IA num ambiente *desktop*). Quando o agente precisa realizar uma tarefa de raciocínio, ele codifica uma requisição e a envia para o servidor que decodifica e processa esta requisição. A resposta é então codificada e enviada de volta ao agente (ver Figura 7.2). Este é o esquema usado por *Jasper* [78] e *Interprolog* [79] para implementar a comunicação entre uma aplicação Java e uma aplicação Prolog em um ambiente *desktop*.

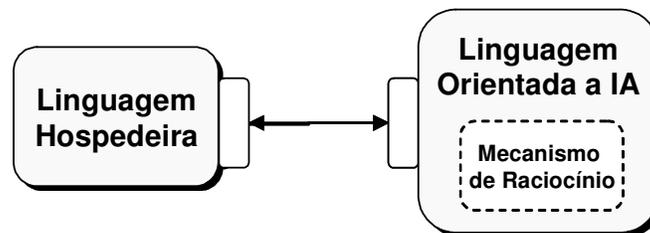


Figura 7.2: Interfaces para interoperabilidade entre sistemas

Infelizmente, esta abordagem requer uma conexão robusta e persistente com o servidor. Estas conexões são muito caras na maioria das redes sem fio disponíveis até hoje. No mais, como já foi discutido em 3.2, o raciocínio embutido é imprescindível em algumas aplicações requerendo reatividade em tempo real, tais como jogos interativos e assistentes de navegação para deficientes visuais.

7.1.3. LINGUAGEM HOSPEDEIRA PARA CONSTRUÇÃO DE APIS

A terceira abordagem é baseada no desenvolvimento de mecanismos para o raciocínio dedutivo numa linguagem hospedeira disponibilizada por uma máquina virtual que garanta uma certa portabilidade. Dentro dessa abordagem, existem duas alternativas.

A primeira alternativa é a construção de tradutores de linguagens, mais especificamente, tradutores de linguagens de programação orientadas a IA (LPOIA) para a linguagem hospedeira. Para uma aplicação escrita numa LPOIA, um tradutor gerará um outro código-fonte traduzido para ser interpretado ou executado pela máquina virtual hospedeira. A aplicação gerada tem implicitamente todas as funcionalidades necessárias para o raciocínio dedutivo e já está pronta para ser usada (ver Figura 7.3). Este é o esquema utilizado por *Minerva* [80], *jProlog* [81], *Prolog Café* [82] e *Jinni* [83] para se implementar a tradução de *Prolog* para Java em ambientes *desktop*.

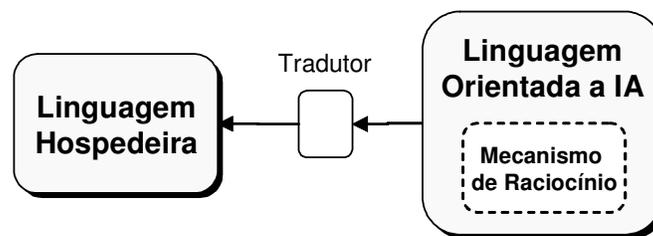


Figura 7.3: Tradutor de linguagens

A segunda alternativa é a construção de um mecanismo de dedução (máquina de inferência) como uma API da linguagem hospedeira (ver Figura 7.4). O sistema desenvolvido na linguagem hospedeira é alimentado por uma base de conhecimento externa usualmente composta de regras de produção e escrita em uma sintaxe arbitrária. Esta base de conhecimentos pode ser tanto interpretada como compilada dependendo de como o desenvolvedor quer carregá-la: em tempo de execução ou estaticamente.

Esta abordagem é normalmente implementada quando se está integrando sistemas de produção com linguagens hospedeiras

orientadas a objetos. Este tipo de integração é chamada EOOPS (*Embedded Object-Oriented Production Systems*) e foi utilizada para o desenvolvimento de CLIPS [16], RAL/C++ [84] para C++, NéOpus [85] para SmallTalk, e JESS [86], JEOPS [08] e OPSJ [87] para Java.

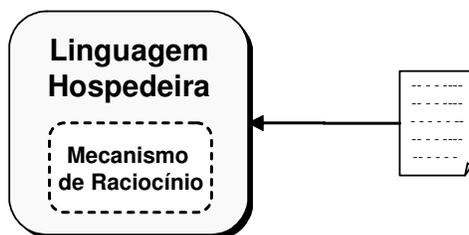


Figura 7.4: Embutindo mecanismo de raciocínio numa linguagem hospedeira

De acordo com a nossa análise, as duas alternativas desta última abordagem são as duas mais viáveis para a implementação de dedução em dispositivos móveis.

Para validar nossa análise, decidimos escolher uma das alternativas e implementá-la. Como já tínhamos experiência no desenvolvimento do JEOPS, um sistema de produção implementado em Java (J2SE), escolhemos a abordagem EOOPS.

7.2. EOOPS EM DISPOSITIVOS MÓVEIS

Dentre as opções de mecanismos de inferência, os sistemas de produção, uma abordagem baseada em regras, são os mais conhecidos e utilizados [88]. A integração entre objetos e regras, que se convencionou chamar de *Embedded Object Oriented Production Systems* (EOOPS), está cada vez mais sendo utilizada pelos desenvolvedores de sistemas dedutivos [89].

7.2.1. SISTEMAS DE PRODUÇÃO

Os sistemas de produção, como qualquer outro sistema de raciocínio dedutivo, é composto de uma linguagem de representação do conhecimento (LCR) e de uma máquina de inferência. Além disso, a

máquina de inferência deve incorporar mecanismos para resolução de conflitos de regras disparáveis.

7.2.1.1. REGRAS

As regras de produção são as mais populares linguagens para representação do conhecimento. Elas são usadas para se representar um conhecimento heurístico sobre o mundo, especificando um conjunto de ações que devem ser realizadas para um determinado estado de uma *base de fatos* que representam um conjunto de verdades sobre o mundo. Uma regra é composta por condições (também conhecida por parte "se" da regra) e por ações (a parte "então" da regra).

7.2.1.2. MÁQUINA DE INFERÊNCIA

O ciclo de execução dos sistemas de produção com encadeamento progressivo possui três passos principais: *unificação*, *resolução de conflitos* e *execução das regras*. No processo da *unificação* o sistema de produção procura casar os fatos presentes na base de fatos com as declarações das regras de modo a tornar verdade os predicados que compõem as condições das mesmas. A cada unificação bem sucedida, o par <regra, fatos> é inserido no *conjunto de conflitos*.

Na *resolução de conflitos* é escolhido um par <regra, fatos> do conjunto de conflitos, de acordo com a política de resolução de conflitos utilizada.

Para a *execução (ou disparo) da regra*, as ações da regra escolhida na etapa anterior são executadas com suas variáveis substituídas pelos fatos que tornaram a regra disparável.

Este ciclo se repete até que não haja mais nenhuma regra disparável. A ação da regra pode alterar a lista de regras disparáveis, através do acréscimo ou da remoção de fatos da memória de trabalho. Estas operações, que se convencionou chamar de "assert" e "retract" respectivamente, são a base da manipulação de fatos da memória de trabalho.

No encadeamento regressivo, também conhecido por *backward-chaining*, ao contrário do progressivo, o encadeamento se dá de um objetivo para as condições necessárias para que este possa ser estabelecido. Em sistemas com encadeamento regressivo, o motor de inferência tenta unificar o lado direito das regras com os objetivos os quais se tenta provar, gerando como novos sub-objetivos as condições definidas no lado esquerdo, até que se atinjam condições atômicas.

7.2.2. EOOPS

A idéia por trás da integração de objetos com os sistemas de produção é juntar as vantagens de sistemas inteligentes, que historicamente estavam restritos a aplicações acadêmicas, com toda a estrutura adquirida pelas linguagens orientadas a objetos durante décadas de pesquisa da Engenharia de Software. Além disso, as relações ontológicas da linguagem orientada a objetos são utilizadas de forma natural pelo sistema.

Nos sistemas EOOPS, os objetos da linguagem são os próprios elementos da memória de trabalho do sistema de produção. As condições das regras são definidas como chamadas a métodos da linguagem (ver Listagem 7.1). Teoricamente todo e qualquer objeto pode ser usado nas regras. De fato, os sistemas baseados em EOOPS não unificam mais cadeias de caracteres, mas sim objetos inseridos em uma base de fatos. A unificação dos elementos das regras e dos objetos torna-se comportamental e não mais estrutural [08].

Nos EOOPS, as condições das regras seguem o modelo de invocação de método $C.m(P_1, \dots, P_n)$ onde m é um método que retorna um valor booleano e C, P_1, \dots, P_n são classes (no caso de m ser estático), interfaces ou instâncias arbitrárias de classes. As entidades P_1, \dots, P_n , ainda podem assumir a forma de métodos que retornam algum valor. Para unificar estas condições com a base de fatos, precisa-se instanciar este modelo de invocação não somente para os fatos que são instâncias de C, P_1, \dots, P_n , mas também dos fatos que são instâncias dos seus descendentes, i.e., que estão abaixo na árvore de hierarquia (as subclasses). Além disso, se C, P_1, \dots ou P_n é uma

interface, o modelo de invocação unificará esta interface com qualquer fato da base de fatos que seja instância de classes que implementam esta interface.

Por exemplo, a regra da Listagem 7.1 tem três condições para disparar suas ações. Cada uma delas é uma expressão Java. As duas primeiras são invocações diretas a métodos. A terceira condição não é uma invocação direta a um método, mas é uma expressão, que em Java, retorna um valor booleano. Na primeira condição (*c.needs(p)*) *needs* é um método da classe *c* que recebe como parâmetro uma entidade *Product* que pode ser uma interface ou uma classe. No caso de *Product* ser uma interface, a unificação casará *p* com instâncias de classes que implementam *Product*. No caso de *Product* ser uma classe, a unificação casará *p* com as instâncias de *Product* e de seus descendentes.

```
rule trade {  
  declarations  
  Salesman s;  
  Customer c;  
  Product p;  
  conditions  
  c.needs(p);  
  s.owns(p);  
  s.priceAskedFor(p) <= c.getMoney();  
  actions  
  s.sell(p);  
  c.buy(p);  
}
```

Listagem 7.1: Exemplo de unificação comportamental

7.2.2.1. VANTAGENS DA INTEGRAÇÃO

A integração de objetos e regras de produção altera a filosofia de trabalho da modelagem e codificação de sistemas de produção. Com efeito, no lugar de simples cadeias de caracteres com uma certa semântica posicional, os fatos da memória de trabalho são substituídos por entidades que são por definição encapsuladas, cuja

estrutura interna não é visível para o mundo exterior, e que podem apresentar diversos relacionamentos com outros objetos.

Esta simples substituição de fatos por objetos traz várias vantagens, assim como algumas desvantagens que discutiremos na próxima seção. As vantagens são tanto ligadas a conceitos de qualidade apresentados pela Engenharia de Software (reusabilidade, modularidade, legibilidade, eficiência, manutenibilidade), como relativas à engenharia de conhecimento (representação natural de relações ontológicas).

A primeira vantagem dessa substituição é a possibilidade de reutilização de objetos incentivada pela filosofia dos sistemas orientados a objetos. Este reaproveitamento é mais interessante quando os objetos têm uma estrutura bem testada (por terceiros) e seus serviços podem ser realmente utilizados.

Uma outra vantagem decorrente da utilização de objetos como elementos da memória de trabalho é que, como certas operações complexas podem ser transferidas das regras para os próprios objetos, as regras do sistema ficam menores, e em menor número, o que contribui para a legibilidade, eficiência e manutenibilidade do sistema como um todo.

Um outro exemplo pode ilustrar melhor este ponto: supondo que um agente tenha que realizar uma busca complexa em um banco de dados, ou mesmo na Internet, para descobrir se *Y* é o pai de *X* – o que pode ser o caso em certos testes de paternidade por DNA – a definição da regra que utiliza o predicado *X.isFatherOf(Y)* não precisaria ser alterada, uma vez que a consulta seria feita pelo próprio objeto. Neste caso, o teste de paternidade poderia ser feito mesmo que para isso o método tivesse de ativar uma nova base de conhecimentos.

Uma terceira vantagem decorrente da união de dois paradigmas distintos, e não somente de regras e objetos, é a possibilidade do programador escolher para cada módulo do sistema o paradigma que melhor convém. Se a integração entre a linguagem e as regras é bem realizada, esta escolha pode ser feita de modo que o sistema apresente um alto grau de modularidade. Por exemplo, na

implementação de um agente ligado na Internet que executa músicas para o seu usuário, a escolha das músicas a serem carregadas de acordo com as preferências do usuário (e possivelmente outros aspectos, como hora de pico de tráfego da rede, preço das tarifas telefônicas) seria implementada através de regras, enquanto que o protocolo de comunicação utilizado na conexão com os servidores de música seria implementado mais facilmente através de comandos da linguagem orientada a objetos. Se o usuário desejar mudar a maneira que ele escolhe suas músicas, ou se o agente decide utilizar um protocolo de comunicação mais eficiente, estas alterações podem ser feitas sem causar impacto no restante do sistema.

As vantagens da integração relativas à engenharia do conhecimento vêm do fato da união entre regras e objetos permitir que o programador separe explicitamente as ontologias do domínio do problema (representadas pelas relações entre os objetos) e suas estruturas de inferência (representadas pelas regras). De fato, através da utilização de objetos, as relações de herança e pertinência podem ser representadas quase que naturalmente. Esta divisão clara entre os conceitos de ontologias e inferência vem sendo defendida por certos autores há algum tempo [08] e é importante para que o programador conheça melhor o sistema que está desenvolvendo.

Todas estas vantagens, especialmente as relativas à qualidade do *software*, nos levam a crer que a integração objetos/regras é interessante, apesar dos problemas que serão apresentados na seção seguinte.

7.2.2.2. PROBLEMAS DA INTEGRAÇÃO

As vantagens obtidas através da integração de objetos com regras de produção têm um custo. O primeiro problema é a notificação de alteração dos objetos [08]. No ciclo de execução de um motor de inferência, existe um passo que verifica se os objetos disponíveis na base de fatos tornam as condições verdadeiras. Em um conjunto de regras onde a inferência é feita em vários passos, podem ocorrer modificações nos objetos que validam ou invalidam outras regras.

O problema está em saber quando os objetos são modificados. De fato, em alguns casos, além dos tipos primitivos, o estado dos objetos é determinado pelo estado de outros objetos. Uma modificação em um desses objetos referenciados não necessariamente ocasionaria uma modificação direta no objeto que faz a referência, uma vez que a mesma não é alterada. Existem dois tipos de soluções para este problema: fazer com que os objetos notifiquem quando da sua real alteração (incluindo as alterações dos objetos referenciados) ou fazer com que o usuário informe explicitamente quando houve uma alteração no objeto.

Um segundo problema é quanto a incerteza de quando utilizar o melhor paradigma, ou seja, decidir o que deve ser feito pelas regras e o que deve ser feito pelos objetos. Esta decisão não é muito clara. Para este problema não existem soluções diretas. Só mesmo a experiência dos desenvolvedores poderá influenciar a decisão.

7.2.2.3. REGRAS PRÉ-COMPILADAS OU INTERPRETADAS

Regras de produção são entidades de representação de conhecimento que não são compreendidas por um computador. Cabe aos sistemas de produção a sua conversão para estruturas que possam ser compreendidas pela linguagem hospedeira. Há basicamente duas maneiras distintas de se implementar esta conversão. Uma delas é a *interpretação* das regras de produção pelo motor de inferência. Desta maneira, o motor é responsável por invocar as operações específicas da linguagem, como operações aritméticas, chamadas a métodos ou criação de novos objetos. Uma segunda abordagem é a *pré-compilação* das regras de produção em estruturas reconhecidas pela linguagem hospedeira.

PRÉ-COMPILAÇÃO DAS REGRAS

Em computação, o termo *compilação* é usado para representar o processo de se transformar um programa escrito em uma determinada linguagem de programação em instruções que possam ser compreendidas pelo computador (código de máquina). A Figura 7.5 ilustra a

compilação do arquivo "Prog.c", que resulta no programa executável "Prog.exe".

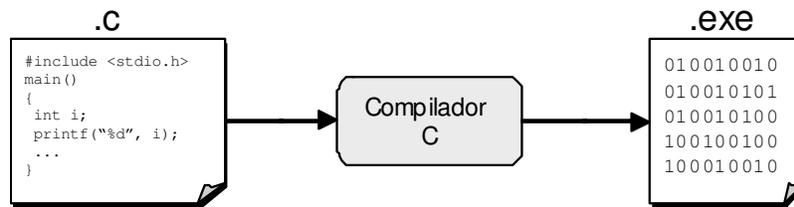


Figura 7.5: Compilação de um programa da linguagem C

A compilação de programas de algumas linguagens, como Java, é diferente. Em vez de gerar um arquivo que pode ser executado diretamente pelo computador, o compilador Java transforma um programa da linguagem em uma série de *bytecodes*, instruções de baixo nível que podem ser *interpretadas* por uma *máquina virtual Java* (ou JVM [90]). Esta característica permite que um programa Java compilado seja executado em diversas plataformas da mesma maneira, garantindo a portabilidade da linguagem. A Figura 7.6 ilustra o processo de compilação de um programa Java.

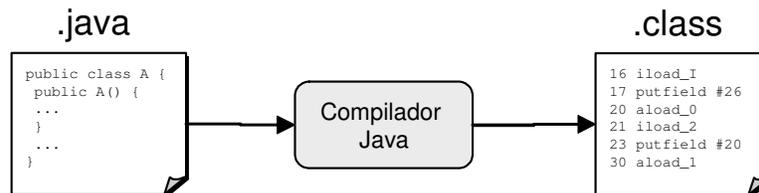


Figura 7.6: Compilação de um programa da linguagem Java

A compilação de regras ocorre de maneira similar à compilação de programas. A diferença é que o resultado da compilação das regras é um código fonte da linguagem hospedeira do sistema de produção, que por sua vez será novamente compilado para poder ser executado, seja diretamente pelo computador ou via uma máquina virtual. Por isso, convencionou-se denominar este processo não de compilação, mas de *pré-compilação*. Após esta etapa, as regras podem então ser compiladas pelo compilador da linguagem hospedeira juntamente com o programa que as utiliza, para que seja gerado o código que será

executado. A Figura 7.7 ilustra este processo para o caso de Java: as regras do arquivo texto onde foram escritas as regras (.rules) são pré-compiladas, gerando o arquivo na linguagem hospedeira (.java) e então compilada com um compilador convencional Java, juntamente com as classes específicas da aplicação usando para isso as classes básicas de infra-estrutura (API Java) e o mecanismo de inferência.

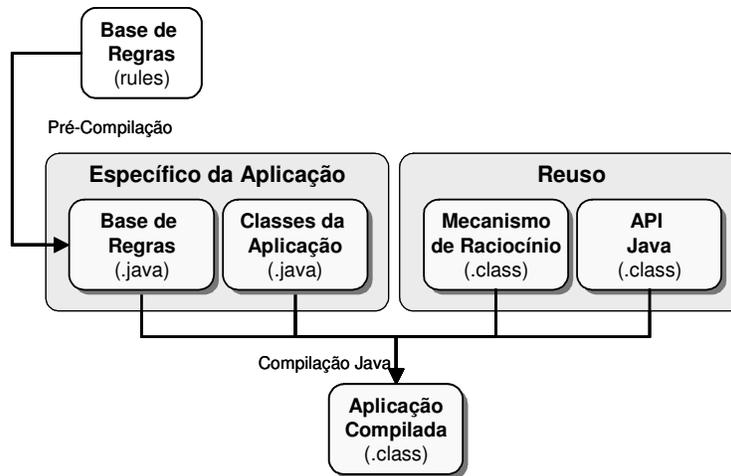


Figura 7.7: Pré-compilação de regras

A pré-compilação das regras é uma estratégia muito utilizada nos sistemas de produção, pois ela facilita a implementação do motor de inferência [08]. De fato, se as regras fossem totalmente interpretadas, o próprio motor teria que realizar diversas tarefas que são realizadas pelo compilador, como a avaliação de expressões aritméticas, a definição e utilização de variáveis. Como o código que será executado é gerado pelo compilador, todas as vantagens obtidas pela otimização de código realizada pelo compilador são mantidas. Uma outra vantagem da compilação é o fato de que o código gerado está num nível bem próximo da linguagem de máquina (quando não for o próprio), e por isso é executado de forma mais eficiente do que se estivesse em um nível mais alto.

A desvantagem da pré-compilação das regras reside no fato de que, uma vez pré-compiladas, a base de regras não pode ser alterada no decorrer da execução da aplicação. Esta limitação, no entanto, é superada em algumas linguagens. *Smalltalk*, por exemplo, permite que

o usuário altere dinamicamente uma classe do sistema, o que torna possível a alteração da base de regras enquanto a aplicação está sendo executada. Algumas máquinas virtuais de Java, como a implementada no ambiente de desenvolvimento *Visual Age*, da IBM [91], também suportam a alteração dinâmica de classes.

INTERPRETAÇÃO DAS REGRAS

A interpretação de regras em um sistema de produção é o processo pelo qual este lê as regras de alguma fonte (console, arquivo texto, etc.) e as armazena em uma estrutura interna (ver Figura 7.8). Na sua execução, o motor de inferência trabalha com esta estrutura, utilizando-a para a unificação das regras, resolução de conflitos e disparo das regras.

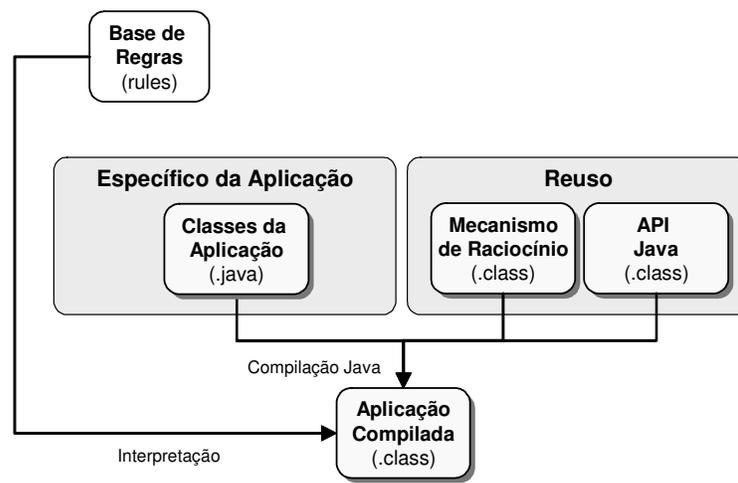


Figura 7.8: Interpretação de regras

A interpretação das regras tem como maior vantagem a possibilidade de se alterar dinamicamente a base de regras do sistema. Como as regras são carregadas em tempo de execução, é possível se realizar uma operação de "reload" das regras para que o comportamento do sistema seja alterado. Uma outra vantagem da interpretação das regras é que o sistema de produção fica independente de um compilador da linguagem para funcionar. De fato, sistemas que implementam a interpretação de regras são ideais para a prototipação rápida de aplicações, pois o usuário não tem que ficar compilando um código fonte a cada alteração que é feita.

No entanto, a interpretação das regras pelo motor de inferência impede que sejam realizadas diversas otimizações de performance que poderiam ser feitas pelo compilador da linguagem hospedeira. Além disso, a implementação de um motor de inferência baseado na interpretação das regras é muito mais trabalhoso, pois o sistema passa a ter que realizar a função do próprio compilador em tarefas que não envolvem o motor de inferência, como criação de variáveis, operações aritméticas, entre outras [08].

7.2.3. EOOPS E J2ME

Como já mencionado na seção anterior, existem vantagens e desvantagens no uso de regras pré-compiladas ou interpretadas. Para a implementação de um EOOPS em dispositivos móveis utilizando J2ME, algumas considerações ainda precisam ser feitas.

Como já mencionado, em um sistema EOOPS pode-se utilizar todos os recursos provenientes da engenharia de software orientada a objetos, em especial a herança de classes. Esta herança garante a unificação de objetos com comportamentos semelhantes, isto é, a unificação de objetos do tipo declarado nas regras com os objetos localizados na base de fatos que herdam o comportamento de tais objetos. Para facilitar o entendimento, considere a regra abaixo que promove um empregado de uma empresa se o seu desempenho é maior que 85%:

***Para todo** empregado do tipo Empregado,
Se o rendimento do empregado é maior que 85%
Então o empregado recebe promoção*

Listagem 7.2: Regra para promoção de um empregado

Suponha agora que na empresa existam vários empregados enquadrados nos mais variados cargos, tais como Engenheiro de Software e Engenheiro de Qualidade. O recurso da herança faz com que a unificação ocorra entre *empregado* (declarado na regra) e os seus

descendentes, no caso Engenheiro de Software e Engenheiro de Qualidade, como ilustrado na Figura 7.9.

Infelizmente J2ME não possui recursos suficientes para promover esta unificação. De fato, para sabermos quais as superclasses dos objetos presentes na base de fatos, precisamos de um recurso somente presente em J2SE, a reflexão Java [92]. A reflexão dá a aplicação o poder de verificar dinamicamente as meta-informações dos objetos, tais como sua superclasse e as interfaces implementadas. A alternativa para o desenvolvedor do sistema de produção é a implementação de tais necessidades.

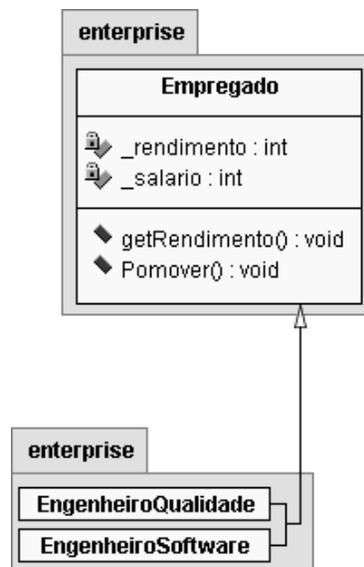


Figura 7.9: Exemplo de Herança

Outro problema relacionado com a reflexão diz respeito à utilização da interpretação de regras. Como já mencionado, as regras são lidas comumente de arquivos texto e armazenadas em estruturas internas entendidas pelo sistema de produção. Esta leitura e transformação de texto em objetos fazem uso da reflexão.

No caso da interpretação da regra por um sistema de produção implementado em J2ME, poderiam até se criar as instâncias das classes *Salesman*, *Customer* e *Product* (da Listagem 7.3) se elas estivessem no *classpath*, isto é, se elas forem classes conhecidas do sistema. No entanto não teríamos como associar as *strings needs(p)*,

owns(p), *priceAskedFor(p)*, *getMoney()*, *sell(p)* e *ebuy(p)* aos seus respectivos métodos, pois J2ME também não provê recursos para visualização dos atributos e/ou métodos dos objetos em tempo de execução. Desta forma, não se pode mapear os métodos aos seus respectivos objetos utilizando-se a interpretação de regras.

```
rule trade {
  declarations
    Salesman s;
    Customer c;
    Product p;
  conditions
    c.needs(p);
    s.owns(p);
    s.priceAskedFor(p) <= c.getMoney();
  actions
    s.sell(p);
    c.buy(p);
}
```

Para todo *Salesman s*,
Customer c e *Product p*...

Se um *c* precisa de um *p*,
um *s* tem um *p* e o *c* tem o
dinheiro pedido pelo *p*...

Então execute os comando:
- *s* vende o *p*
- *c* compra o *p*

Listagem 7.3: Regra JEOPS

Este problema é resolvido quando a pré-compilação de regras é utilizada, pois as regras geradas pelo pré-compilador são classes Java. Nenhum mapeamento tem que ser feito para associar cadeias de caracteres com métodos reais.

7.3. FERRAMENTAS A SEREM REUSADAS

Uma vez definido o tipo de sistema de raciocínio dedutivo a ser usado, assim como a linguagem de implementação, faz-se necessária a implementação em si. Incentivados pelos conceitos de reusabilidade do paradigma da orientação a objeto, que prega o aumento da produtividade quando componentes já implementados são reusados, decidimos reutilizar uma ferramenta já desenvolvida em J2SE.

Com efeito, decidimos identificar os principais sistemas que seguem os conceitos de EOOps, analisá-los e possivelmente reutilizá-

los na produção de um EOOPS embutido em dispositivos móveis. Nesta seção serão apresentadas as ferramentas analisadas: JESS e JEOPS. Além dessas duas ferramentas existe outra ferramenta chamada OPSJ [87], que parece apresentar os mesmos princípios gerais de JEOPS, mas no entanto, não conseguimos fazer uma análise mais detalhada devido à não disponibilidade do OPSJ no mercado brasileiro.

7.3.1. JESS

No início de 1997, o pesquisador *Ernest J. Friedman-Hill*, do laboratório *Sandia da Califórnia*, criou um clone de CLIPS [16] para a linguagem Java. Foi a primeira proposta divulgada de integração entre sistemas de produção e a linguagem, que na época já aparecia como uma idéia viável, principalmente pelo apelo da Internet.

7.3.1.1. APRESENTAÇÃO

A idéia de JESS (*The Java Expert System Shell*) [86] é a conversão de um subconjunto (inicialmente) de CLIPS para Java, mantendo a mesma filosofia do sistema desenvolvido pela NASA. No princípio, o sistema apresentava diversas limitações: não suportava naturalmente a herança de objetos e apresentava diversos *bugs* documentados, entre outras. Entretanto, a versão atual (6.0) já resolveu grande parte destes problemas, possibilitando, inclusive, a utilização de raciocínio com encadeamento regressivo, serviço não suportado por nenhum dos outros sistemas estudados.

A popularidade de JESS cresceu grande parte devido à base de usuários de CLIPS que desejavam migrar de soluções baseadas em C/C++ para Java e utilizar os serviços oferecidos por esta linguagem. Existem milhares de usuários do sistema, que contam com uma lista de discussão [93] relativamente movimentada, fazendo com que o suporte ao uso deste sistema seja bastante efetivo.

Assim como CLIPS, JESS define uma linguagem chamada COOL [86] para ser usada na criação de aplicações, e não apenas de regras.

Comandos de criação de interfaces gráficas, comunicação e de entrada/saída são disponibilizados nesta linguagem. Pode ser desenvolvida uma aplicação inteira dentro do *shell* de JESS.

JESS pode utilizar como elementos de sua memória de trabalho objetos Java. No entanto, existe uma restrição para os objetos Java que podem ser usados. Como não existe nenhuma forma explícita de se avisar ao sistema quando um objeto é modificado (como o *modified* de NéOpus ou o *modify* de RAL/C++), os objetos devem ser capazes de avisar quando uma de suas propriedades é alterada.

7.3.1.2. CRÍTICA

Apesar de ser o primeiro sistema de produção definido para Java, JESS sofre dos mesmos problemas de CLIPS: basicamente a baixa integração do sistema com a linguagem hospedeira [08]. Ainda existe a separação dos mundos de objetos Java e o de objetos (ou fatos) JESS, e a sintaxe utilizada na definição das regras em nada se assemelha à da linguagem hospedeira, fazendo com que o usuário, tipicamente um programador Java, tenha que passar um tempo maior no entendimento do sistema (ver).

```
( defrule library-rule-1
  ( book (name ?X) (status late) (borrower ?Y) )
  ( borrower (name ?Y) (address ?Z) )
  =>
  ( send-late-notice ?X ?Y ?Z )
)
```

Listagem 7.4: Regra JEOPS

Esta regra pode ser traduzida para pseudo-código da Listagem 7.5:

```
Regra #1:
  Se
    Existe um livro em atraso, com nome "X", emprestado a
    alguém chamado "Y" e cujo endereço é "Z"
  Então
```

Envie uma nota de atrso para "Y" no endereço "Z" sobre o livro "X".

Listagem 7.5: Regra JEOPS

A necessidade que os objetos sejam capazes de avisar sobre modificações ocorridas em suas propriedades também limita a utilização do sistema. De fato, o sistema fica impossibilitado, por exemplo, de utilizar classes definidas por terceiros, uma prática bastante comum no desenvolvimento orientado a objetos.

A eficiência de JESS também é um pouco prejudicada por ser um sistema interpretado pois não toma proveito dos compiladores JIT [94] que já se tornaram padrão para Java. A utilização de um mundo de objetos a parte dos objetos de Java também causa problemas para JESS. Uma *String* em JESS consiste em um objeto da classe *jess.Value* que contém uma *String* de Java. Com isso, uma chamada ao método *substring* em Java cria um objeto como resultado, enquanto que em JESS dois objetos são criados. O número 5 em Java é um tipo primitivo - pode ser armazenado em um registro - enquanto que em JESS é um objeto que contém o número 5 nele. Com isso, se em Java a operação "2 + 3" não cria nenhum objeto, (+ 2 3) em JESS cria um. Como a criação de objetos é uma operação bastante cara em Java, a performance do sistema fica prejudicada.

7.3.2. JEOPS

JEOPS (*Java Embedded Objects Production System*) é um sistema de produção implementado em Java (J2SE) desenvolvido no Centro de Informática (CIn) da Universidade Federal de Pernambuco em 1999 e vem sofrendo constantes aprimoramentos desde então.

7.3.2.1. APRESENTAÇÃO

A idéia de JEOPS é estender a linguagem Java com regras de produção tendo o máximo de uniformidade na integração de regras e objetos, i.e., fazer com que um desenvolvedor Java não tivesse muitos problemas em escrever uma regra JEOPS. O resultado foi a

especificação de uma sintaxe de regras onde declarações Java podem ser usadas nas declarações das regras e expressões Java usadas nas condições e ações das regras, como ilustrado na Listagem 7.1.

Para o problema da modificação dos objetos, JEOPS dispõe da função *modified* (como o *modified* de NéOpus ou o *modify* de RAL/C++) que avisa à base que o objeto passado como parâmetro foi alterado. Esta função deve ser usada explicitamente nas regras.

Na primeira tentativa, JEOPS foi implementado como um interpretador de regras, i.e., as regras eram escritas em um arquivo texto que era distribuído junto com a aplicação. Esta por sua vez também carregava o núcleo do JEOPS que interpretava as regras. Na segunda tentativa (segunda versão), o JEOPS passou a ser distribuído em dois ambientes distintos, um para pré-compilação de regras e outro para a execução. Desta vez as regras seguem o processo de pré-compilação onde o arquivo texto é transformado em classes Java.

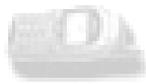
7.4. CONCLUSÃO

Neste capítulo foi discutido como proporcionar a um sistema capacidades de raciocínio, mais especificamente como se introduzir funcionalidade de um sistema de raciocínio dedutivo em um dispositivo móvel. Dentre as opções analisadas para proporcionar estas funcionalidades, a utilização de uma linguagem hospedeira provida por uma máquina virtual mostrou-se ser uma opção viável. De fato, com o advento de J2ME esta opção ficou ainda mais interessante.

Resolvemos utilizar os conceitos de EOOPS para a implementação do sistema dedutivo na forma de um sistema de produção, técnica muito usada ultimamente [89]. Analisamos, então as principais limitações de J2ME que trariam um impacto direto à implementação de um sistema de produção.

Depois de avaliados estes impactos, analisamos dois sistemas de produção implementados na plataforma Java, mas especificamente na

versão *Standard* (J2SE). Analisadas as vantagens e desvantagens de cada sistema, o sistema JEOPS se mostrou o mais compatível com as definições dos sistemas EOops e por isso foi escolhido, dentre outros motivos, para ser reutilizado.



KEOPS

Este capítulo descreve detalhadamente o trabalho de implementação do componente de raciocínio dedutivo para dispositivos móveis, o KEOPS. O KEOPS é o acrônimo para "*Embedded Objects Production Systems*", onde o "K" significa algo desenvolvido para a máquina virtual J2ME, a KVM.

O KEOPS é uma extensão da arquitetura de JEOPS (implementação de um sistema de produção em J2SE), onde os agentes implementados em J2ME e rodando nos dispositivos móveis podem processar localmente tarefas envolvendo inferências dedutivas.

A seguir, o JEOPS será detalhado com suas características, arquitetura e funcionalidades. Depois disto, o trabalho de extensão será explicitado com suas principais dificuldades e soluções. Por fim, serão mostrados alguns exemplos de uso do KEOPS.

8.1. JEOPS

JEOPS (*Java Embedded Objects Production System*) [08] é um sistema de produção implementado em Java (J2SE) cuja primeira versão foi desenvolvida em 1999, fruto de um trabalho de graduação de Carlos Santos Figueira Filho, um aluno do Centro de Informática (CIn) da Universidade Federal de Pernambuco (UFPE). No seu curso de

mestrado, Carlos aperfeiçoou o JEOPS fazendo, entre outras coisas, com que suas regras passassem a ser compiladas em objetos Java.

8.1.1. SINTAXE REGRAS

JEOPS, além de obedecer a todos os conceitos de EOOps, tem como princípio a facilidade de uso e o respeito às características do paradigma orientado a objetos, o que é conseguido com uma uniformidade de integração entre as regras e a linguagem hospedeira. Por este motivo, a sintaxe das regras JEOPS é bastante parecida com a sintaxe de Java.

As regras JEOPS são separadas em três partes: declarações de variáveis (*declarations*, *localdecl*), condições (*conditions*) e ações (*actions*). Regras JEOPS são organizadas dentro de *base de regras*, tal como métodos em Java são organizados dentro de classes. Uma base de regra contém regras, mas também pode conter métodos e declarações de variáveis, como qualquer classe de Java. Outra característica importante das regras JEOPS é que elas podem fazer referência a qualquer objeto Java do mesmo modo que métodos Java fazem referências a objetos. Desse modo, quando a base de regras for utilizada, os objetos utilizados nas regras já devem ter sido definidos.

A Listagem 8.1 ilustra um trecho da definição das classes *Pessoa* e *Objetivo*, usadas na regra apresentada na Listagem 8.2.

<pre>public class Pessoa { private String nome; private Pessoa pai, mae; public String getNome() { ... } }</pre>	<pre>public class Objetivo { private boolean ativo = true; private Pessoa alvo; public Objetivo(Pessoa p) { ... } }</pre>
--	---

Listagem 8.1: Classes *Pessoa* e *Objetivo*

```
ruleBase Familia  
    rule encontraAncestrais {
```

```

declarations
    Pessoa p;
    Objetivo o;
localdecl
    Pessoa pai = p.getPai();
    Pessoa mae = p.getMae();
conditions
    p == o.getAlvo();
    o.estaAtivo();
actions
    o.desativa();
    System.out.println(
        pai.getNome() + " e " + mae.getNome() + " são ancestrais"
    );
    assert(new Objetivo(pai));
    assert(new Objetivo(mae));
}
}

```

Listagem 8.2: Base que contém a regra encontraAncestrais (JEOPS)

Nesta regra, *encontraAncestrais*, o campo das declarações contém as variáveis que serão unificadas com os objetos da memória de trabalho. No caso, a regra precisa de um objeto da classe *Pessoa* (*p*) e um da classe *Objetivo* (*o*). O campo das declarações locais é utilizado para abreviações de expressões que sejam muito utilizadas na regra. No exemplo, toda ocorrência livre dos identificadores *pai* e *mae* serão substituídas pelas expressões *p.getPai()* e *p.getMae()*, respectivamente.

As condições da regra (*conditions*) podem ser qualquer expressão booleana de Java. Em geral, tratam-se de comparações ou chamadas a métodos que retornam valores booleanos. No exemplo da Listagem 8.2, a regra estará disparável para todo par de objetos *o* e *p* das classes *Objetivo* e *Pessoa* respectivamente, que tornem as duas expressões (*o.getAlvo() == p* e *o.estaAtivo()*) verdadeiras.

Finalmente, as ações da regra (*actions*) podem ser quaisquer expressões de Java. Existem também as ações para manipulação dos objetos da base de fatos (*assert*, *retract*, *flush* e *modified*). Tudo o

que pode aparecer como um corpo de método Java pode aparecer neste campo. Na regra da Listagem 8.2, por exemplo, são feitas chamadas a métodos, criação de objetos e impressão de mensagens na saída padrão.

Além disso, através dos comandos *assert* e *retract*, objetos podem ser inseridos e removidos da base de fatos, respectivamente. Modificações nos objetos devem ser informadas ao sistema pelo usuário, através da chamada ao método *modified*. Este problema da modificação transitiva de objetos é contornado, da mesma forma que *NéOpus*, fazendo com que o usuário declare todas as variáveis que, uma vez modificadas, possam fazer a regra alterar de estado (de disparável para não disparável).

8.1.2. MÁQUINA DE INFERÊNCIA

A máquina de inferência JEOPS, além de proceder a unificação das regras com os objetos da base de fatos, possui um mecanismo para resolução de conflitos. Na resolução de conflitos, o sistema deve decidir qual dos pares *<regra,fatos>* presentes no conjunto de conflitos deve ser o escolhido para executar. JEOPS possui uma série de conjuntos de conflitos com políticas diferentes para resolução de conflitos, entre elas:

- **Política Padrão:** qualquer instância da base de fatos pode ser disparada.
- **Política LRU:** a regra menos usada é escolhida para o disparo.
- **Política MRU:** a regra mais usada é escolhida para o disparo.
- **Política Natural:** regras disparadas não podem ser mais disparadas para um mesmo objeto.
- **Política OneShot:** regras disparadas não podem mais ser disparadas, mesmo para objetos distintos.

- **Política de Prioridade:** as regras definidas primeiramente na base de regras têm prioridade.

Escolhida uma regra para disparar, o sistema deve então, proceder a substituição das variáveis da regra pelos fatos que tornaram a regra disparável e executar as ações da regra (*actions*).

8.1.3. PROCESSO DE DESENVOLVIMENTO

A arquitetura de JEOPS utiliza o esquema de pré-compilação de regras como discutido em 7.2.2.3. Esta arquitetura é dividida em dois ambientes: um de compilação e outro de execução.

As regras são escritas em arquivos de texto puro e traduzidas (pré-compiladas) em classes Java (.java) pelo ambiente de compilação. O arquivo gerado é compilado juntamente com as classes da aplicação por um compilador convencional de Java. As classes compiladas são então distribuídas junto com o ambiente de execução de JEOPS que processará as requisições da classe representando a base de regras (ver Figura 8.1).

Em termos mais práticos, o processo de desenvolvimento de uma aplicação JEOPS pode ser descrito pelos seguintes passos:

01. Pré-compilação de regras:

```
java jeops.compiler.Main <base de regras>
```

02. Compilação das classes:

```
javac <classes da aplicação>
```

03. Uso do ambiente de execução:

```
// Criar um objeto da classe gerada na pré-compilação da  
// base de regras, passando como parâmetro o conjunto de  
// conflito com a devida política de resolução de conflitos  
RuleBase base = new RuleBase(  
    new PriorityConflictSet()  
);  
  
// Inserir os objetos na base com a função 'assert'  
kb.assert(f);
```

```
// Executar a base
kb.run();
```

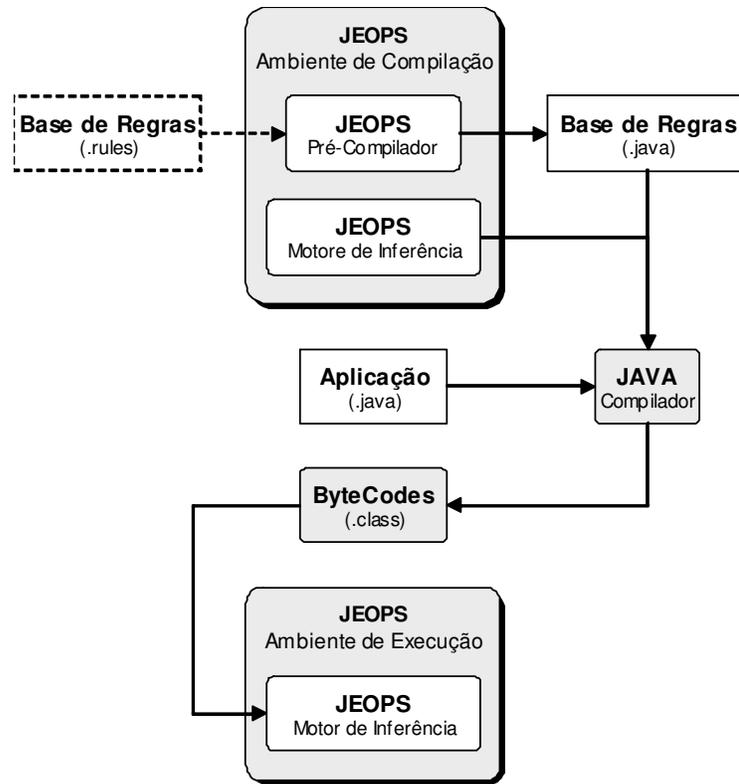


Figura 8.1: Processo de desenvolvimento de uma aplicação JEOPS

8.2. TRABALHO DE ADAPTAÇÃO: KEOPS

Depois de anunciadas as características principais de JEOPS, detalharemos o processo de adaptação e extensão da sua arquitetura para a produção do KEOPS, a segunda grande contribuição deste trabalho.

O processo de extensão da arquitetura JEOPS foi feito em três passos. Inicialmente, inspecionamos o código dos dois ambientes do JEOPS para detectar o uso de funcionalidades Java não suportadas por J2ME. Depois de identificadas as incompatibilidades, listamos as alternativas para solucionar cada uma delas e finalmente, avaliamos

as vantagens e desvantagens de cada alternativa para então escolhermos a melhor e implementá-la.

8.2.1. DIFICULDADES

No processo de identificação das incompatibilidades entre J2SE e J2ME utilizadas por JEOPS, tínhamos que analisar as partes de JEOPS expostas a J2ME. Estas partes são as classes que implementam o ambiente de execução de JEOPS, que vão estar dentro dos dispositivos e as classes geradas pelo ambiente de compilação de JEOPS.

Identificamos que o ambiente de compilação preserva a compatibilidade na compilação de regras, i.e., para bases de regras que preservam em suas declarações e ações a compatibilidade com J2ME, o compilador JEOPS gera classes Java também compatíveis com J2ME. Conseqüentemente, o ambiente de compilação JEOPS não precisou ser alterado e pode ser utilizado como está.

O próximo passo é investigar o ambiente de execução. Identificamos incompatibilidades de quatro tipos diferentes:

- Ausência de métodos em algumas classes, mas que possuem suas funcionalidades implementadas por outros métodos;
- Ausência de constantes em algumas classes;
- Ausência de algumas classes;
- Ausência de métodos em algumas classes, mas que não possuem uma implementação alternativa possível em J2ME;

A primeira e a segunda incompatibilidade foram facilmente resolvidas com soluções alternativas providas pelo próprio J2ME. Os métodos ausentes foram substituídos por suas implementações alternativas e as utilizações das constantes ausentes foram substituídas por literais equivalentes.

As classes ausentes felizmente tinham uma implementação compatível com J2ME e foram incluídas juntamente com o pacote do ambiente de execução do JEOPS.

Os métodos ausentes sem implementação alternativa foram os métodos que implementavam os recursos de identificação de *meta-dados* de classes, mais especificamente os métodos de identificação das superclasses e das interfaces implementadas, presentes na classe `java.lang.Class`. O ambiente de execução do JEOPS faz uso desses recursos para implementar a já mencionada unificação comportamental (ver 8.1.2 e 7.2.3). Na próxima seção serão discutidas as alternativas para a solução desta incompatibilidade.

8.2.2. UNIFICAÇÃO COMPORTAMENTAL SEM REFLEXÃO

O processo de unificação de regras e objetos necessita que o ambiente de execução conheça o tipo dos objetos da base de fatos, ou seja, sua árvore de hierarquia. Por exemplo, suponha a pré-compilação de uma base de regras *B* que possui a regra *R* que por sua vez possui a declaração de uma classe *C* e de uma interface *I*. Como estas entidades (*C, I*) serão unificadas com os objetos da base de fatos, o ambiente de execução de JEOPS precisa saber quais os descendentes de *C* e as classes que implementam *I*.

Uma vez que J2ME não tem recursos suficientes para a extração dinâmica dessas informações, a solução seria persistir estas informações na forma de um arquivo e disponibilizá-lo com a aplicação. A sintaxe desse arquivo seria algo parecido com a Listagem 8.3 que utiliza o exemplo supracitado (*C1* e *C2* são subclasses de *C*. *I1* e *I2* são interfaces que estendem a interface *I*. *C3* e *C4* são classes que implementam a interface *I*).

```
Entidade; Subclasses; Classes que Implementam a Entidade
C; [C1,C2]; []
I; [I1,I2]; [C3,C4]
```

Listagem 8.3: Exemplo de arquivo com as meta-informações.

Uma vez produzido, este arquivo seria distribuído juntamente com a aplicação na forma de recurso (assim como as imagens). O ambiente de execução KEOPS (modificação do de JEOPS) será o responsável pela leitura e processamento das informações contidas neste arquivo.

Para a sua produção e solução do problema da unificação comportamental no KEOPS, enxergamos três abordagens:

- Requisição de ajuda direta do usuário
- Modificação do ambiente de compilação JEOPS
- Adição de um passo de pós-processamento depois da compilação do código da aplicação em *bytecodes*

A primeira abordagem coloca sob a responsabilidade do usuário a declaração explícita das classes que precisam ser consideradas para o modelo de invocação das condições das regras da base como discutido em 7.2.2 (classes descendentes e que implementam as entidades declaradas). Nesta primeira abordagem, o arquivo com as informações necessárias será escrito diretamente pelo usuário (ver Figura 8.2).

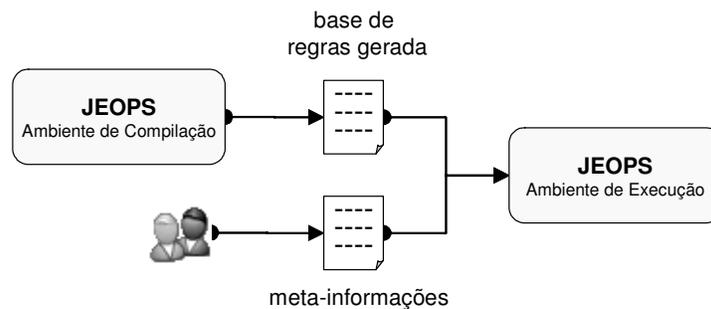


Figura 8.2: O usuário informa explicitamente os meta-dados das classes.

Na segunda abordagem, a produção do arquivo com as informações necessárias para o ambiente de execução estará a cargo do compilador JEOPS que procederá a geração não de um, mas dois arquivos (ver Figura 8.3). Por ainda estar num ambiente suportado por uma máquina virtual J2SE, o compilador JEOPS ainda pode utilizar os recursos de reflexão para extrair as informações necessárias.

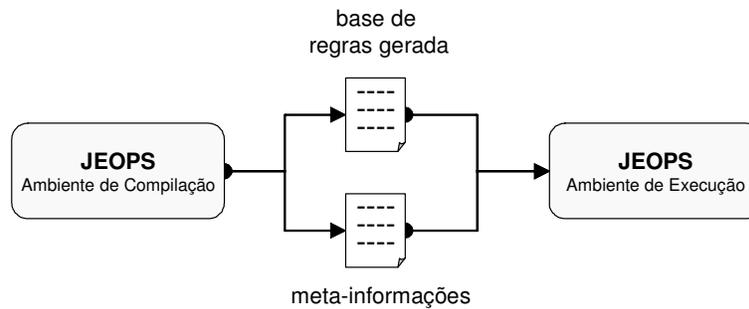


Figura 8.3: O compilador JEOPS gera as informações necessárias.

Na terceira abordagem, a produção do arquivo com as informações necessárias estará sob responsabilidade de um pós-processador que trabalha com *bytecodes*. De fato, este pós-processador por também ainda estar num ambiente suportado por uma máquina virtual J2SE, pode se valer das funcionalidades da reflexão para extrair as informações necessárias dos *bytecodes* da aplicação compilada, incluindo a classe gerada da base de regras (ver Figura 8.4).

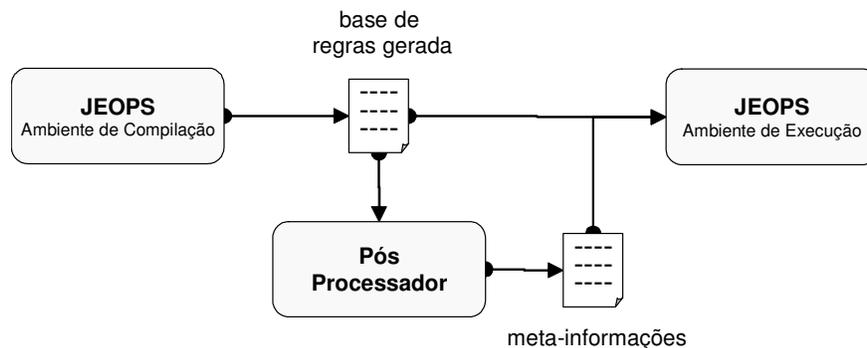


Figura 8.4: Um pós-processador gera as informações necessárias.

Dadas as três opções, tínhamos que escolher a que trouxesse menos desvantagens para os desenvolvedores de aplicações JEOPS. Das três opções, rejeitamos de imediato a primeira, uma vez que ela isenta os usuários da abstração provida pela herança. Com efeito, esta decisão aproximaria o KEOPS mais de um sistema de produção embutido em uma linguagem imperativa do que de um genuíno EOOPS.

Entre as duas opções automatizadas, preferimos a última opção por três razões principais:

- Ela separava organizadamente dois assuntos ortogonais em diferentes componentes: a solução do problema da ausência de recursos para a reflexão (próprio de J2ME) e a pré-compilação de regras em classes Java.
- Ela é mais segura de implementar, uma vez que não é preciso modificar o ambiente de compilação JEOPS.
- Ela é mais simples e rápida de implementar, uma vez que trabalhando no nível de *bytecodes*, o pós-processador pode simplesmente usar os métodos da reflexão Java para gerar as informações necessárias para o ambiente de execução.

Para implementar esta opção, precisaríamos não somente construir o pós-processador, mas modificar o ambiente de execução para que ele acessasse as informações geradas em vez de usar os métodos provenientes da reflexão (*getInterfaces()* e *getSuperClasses()*) (ver Figura 8.5).

De fato, a classe *Serializer* mostrada na Figura 8.6 em um diagrama de classes, implementa esta extração. Esta classe recebe como entrada uma classe do tipo *AbstractKnowledgeBase* (superclasse de todas as bases de regras geradas pelo compilador JEOPS), captura todas as declarações feitas nas regras da base e para cada uma encontra as subclasses e/ou interfaces implementadas através da classe *RTSI*. Para cada tipo de declaração é gerado um objeto da classe *ClassInfo*, que guarda as informações necessárias e é armazenado em um objeto da classe *ClassInfoHash*. Com todas as informações coletadas, a classe *Serializer* pode gerar o arquivo com o método *store()*.

As modificações no ambiente de execução consistiam em escrever um módulo para ler as meta-informações das classes armazenadas no arquivo gerado pelo pós-processador e substituir as invocações dos métodos da reflexão por invocações a métodos de um novo módulo que faz referência às informações lidas pelo primeiro módulo.

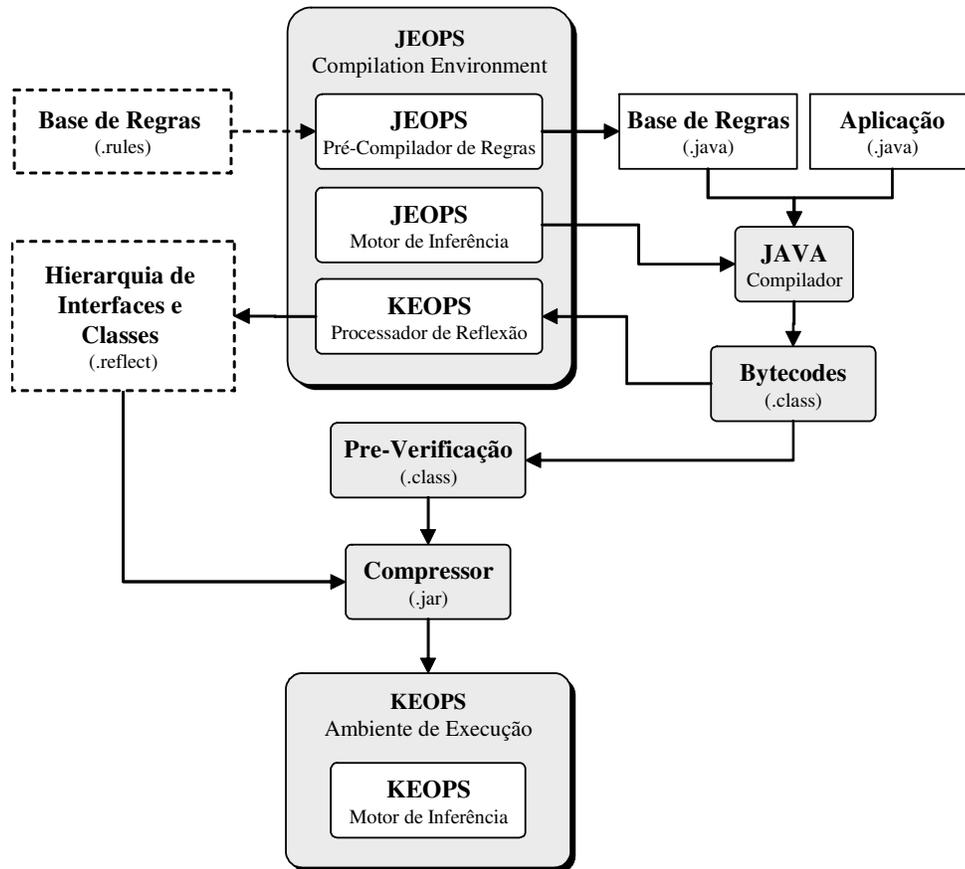


Figura 8.5: Processo de desenvolvimento de uma aplicação KEOPS

Com a implementação da terceira opção, o desenvolvimento de uma aplicação KEOPS passa a ter um passo adicional em comparação ao desenvolvimento de uma aplicação JEOPS. A Figura 8.5 mostra o processo de desenvolvimento de uma aplicação KEOPS onde as regras são escritas na linguagem das regras e pré-compiladas pelo pré-compilador JEOPS. A classe gerada é então compilada juntamente com as classes da aplicação e do ambiente de execução por um compilador Java convencional. Com as classes compiladas, o pós-processador extrai as meta-informações necessárias para o ambiente de execução KEOPS. As classes da aplicação compiladas são submetidas a uma pré-verificação (processo comum para aplicações desenvolvidas em J2ME) e uma compressão JAR (ferramenta disponível no ambiente de desenvolvimento Java - JDK). As informações extraídas são comprimidas juntamente com as classes da aplicação. O arquivo

comprimido é distribuído e a aplicação instalada nos dispositivos onde atuará o ambiente de execução KEOPS.

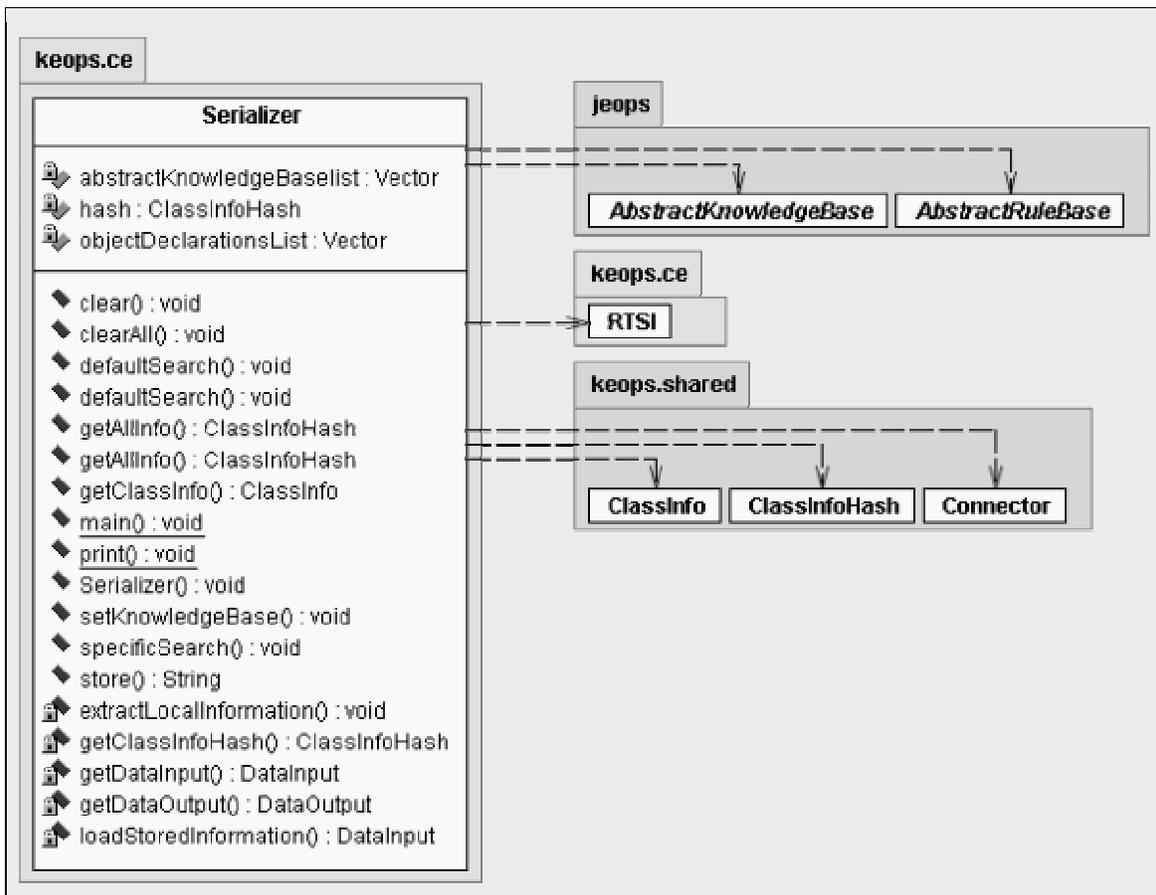


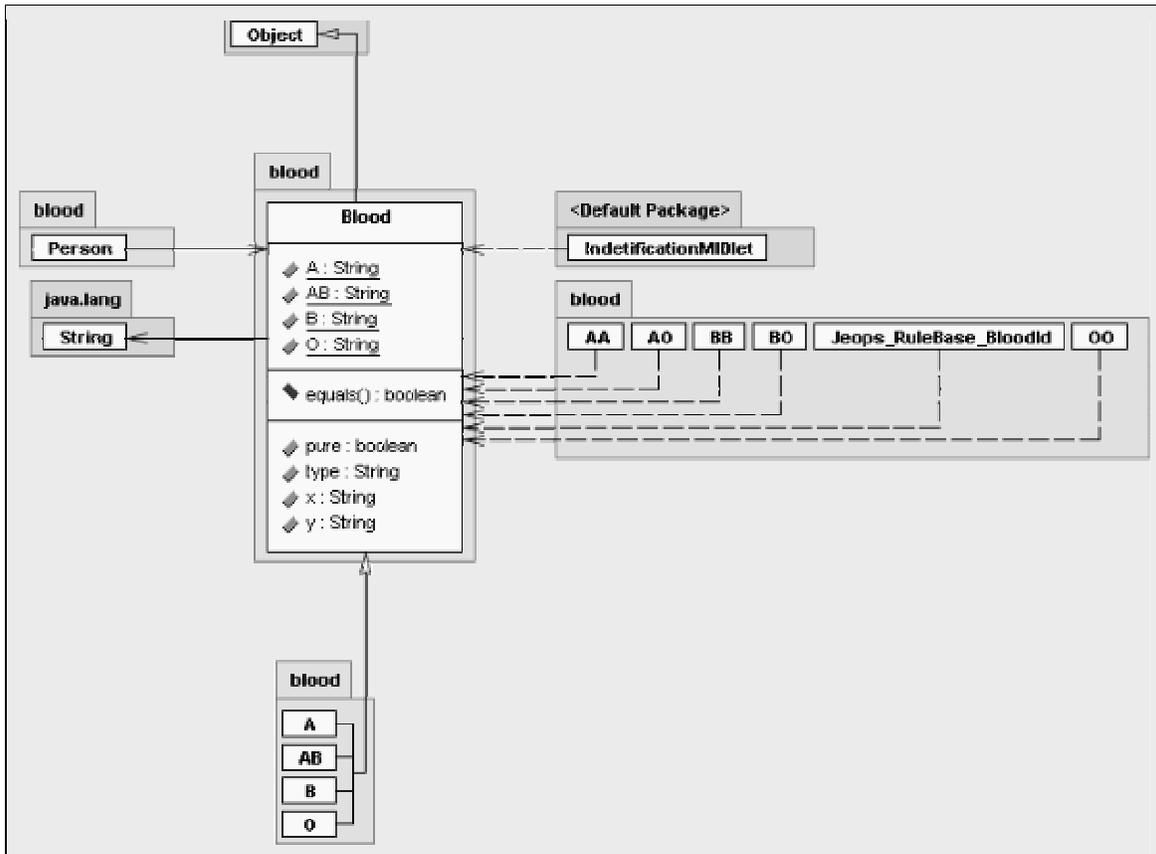
Figura 8.6: Pós-Processador (Classe Serializer)

8.3. FUNCIONAMENTO

Como já mencionado, o desenvolvimento de uma aplicação KEOPS é igual ao desenvolvimento de uma aplicação JEOPS com exceção do passo de pós-compilação. Para visualizar de forma prática este desenvolvimento, veremos nesta seção um exemplo da sua utilização.

8.3.1. EXEMPLOS

Para expor o funcionamento do KEOPS, utilizaremos uma aplicação para identificar as possíveis combinações do tipo sanguíneo de um decendente de dois indivíduos. A **Listagem 8.5** ilustra a classe *Blood*, a **Listagem 8.6** uma das subclasses da classe *Blood*, a **Listagem 8.7** ilustra a base de regras *BloodRules*.



Listagem 8.4: Diagrama de Hierarquia.

```

package blood;

public class Blood {

    private boolean pure;

    private String x;

    private String y;

    public boolean isPure() { ... }

    public boolean equals (Blood b) { ... }
  
```

```

    public String getType() { ... }

    public void setPure(boolean pure) { ... }

    public void setType(String type) { ... }

    public void setY(String y) { ... }

    public void setX(String x) { ... }

}

```

Listagem 8.5: Classe Blood.

```

package blood;

public class AO extends A {

    public AO() {

        super.setX(Blood.A);

        super.setY(Blood.O);

        super.setPure(false);

    }

}

```

Listagem 8.6: Classe AO.

```

package blood;
public ruleBase BloodRules

    rule ruleAABBAABB {

        declarations

            Person son;

            Blood b1;

            Blood b2;

        conditions

            b1 != b2;
    }
}

```

```

        son.getProbabilities().isEmpty();

        !b1.isPure();

        !b2.isPure();

        b1.equals(b2);

        actions

        son.addProbability(new AO(), "25%");

        son.addProbability(new AB(), "50%");

        son.addProbability(new BO(), "25%");

    }

    ...

}

```

Listagem 8.7: Base de regras para o exemplo de identificação do tipo sanguíneo (apenas uma regra).

Após ser pré-compilada a base de regras *BloodId* é representada por uma classe chamada *BloodId.java*. Depois da base ser compilada junto com as classes da aplicação e o ambiente de execução de KEOPS, o pós-processador extrai as meta-informações e as persiste num arquivo a ser lido pelo ambiente de execução KEOPS. O arquivo gerado possui as informações ilustradas na Listagem 8.8.

```

blood.Person|[]|[]|
blood.Blood|[blood.A, ..., blood.BO, blood.O, blood.OO]|[]|

```

Listagem 8.8: Informações do arquivo gerado.

Para que o arquivo seja gerado, é necessária a execução das linhas de comando ilustradas na Listagem 8.9.

```

java jeops.compiler.Main BloodId.rules
javac *.java
java keops.ClassInfoExtractor BloodId

```

Listagem 8.9: Comandos para a geração do arquivo com informações de reflexão.

A **Listagem 8.10** mostra a utilização em Java da base de conhecimentos, a *BloodId*. É construído um objeto *BloodId* e inserido

na base através de um `assert()`. De posse do objeto `BloodId`, a base é então posta para executar com o método `run()` após os devidos objetos serem inseridos na base.

```
BloodId kb = new BloodId();
b.addRuleFireListener(new RuleFireListener() {
    public void ruleFiring(RuleEvent e) {
        int i = e.getRuleIndex();
        String name =
            e.getKnowledgeBase().getRuleBase().getRuleNames()[i];
        System.out.println("Firing rule " + name);
    }

    public void ruleFired(RuleEvent e) {}
});

try {
    kb.flush();
    kb.assert(son);
    kb.assert(new AB());
    kb.assert(new AB());
    kb.run();
    kb.flush();
} catch (Exception e) {
    System.out.println("e = " + e);
    e.printStackTrace();
}
```

Listagem 8.10: Exemplo da utilização da base de fatos `BloodRules`

8.4. RESULTADOS

Para validar as implementações feitas no desenvolvimento do KEOPS, adaptamos os exemplos disponíveis no pacote do JEOPS submetendo-os ao processo de desenvolvimento do KEOPS. Os exemplos foram testados nos mais variados emuladores de dispositivos (e.g. *Sun Wireless Toolkit*, *MotoSDK*, *3Com PalmOS Emulator*), em dispositivos reais disponíveis no mercado, tais como *3Com Palm m100* e *Vx*, *Motorola i85s*, *Siemens SL45i* e em dispositivos ainda não

disponíveis no mercado, cedidos gentilmente pela equipe de desenvolvimento da Motorola no Brasil (BDC) através do BDC/CESAR KJAVA Program.

Os exemplos foram acrescidos de contagens de consumo de memória no qual alguns resultados estão expostos nas tabelas abaixo:

Factorial	
Número	Memória
1	10300 bytes
5	29100 bytes
10	53700 bytes
15	78300 bytes
20	103100 bytes
25	127800 bytes
30	152300 bytes

Tabela 8.1: Consumo de memória do exemplo *Factorial* executando no emulador *Sun J2ME Wireless Toolkit*

Fibonacci	
Número	Memória
1	29600 bytes
5	95700 bytes
6	140600 bytes
7	220000 bytes
8	344900 bytes

Tabela 8.2: Consumo de memória do exemplo *Fibonacci* executando no emulador *Sun J2ME Wireless Toolkit*

Hanói Towers		
Discos	Memória	Tempo
1	29400 bytes	400ms
2	38100 bytes	470 ms
3	59300 bytes	550 ms
4	101800 bytes	610 ms
5	186900 bytes	671 ms
10	Swap bytes	11500 ms

Tabela 8.3: Consumo de memória e tempo de execução do exemplo *Hanói Towers* executando no emulador *Sun J2ME Wireless Toolkit*



9.

CONCLUSÃO

O crescimento do mercado e conseqüentemente da indústria de dispositivos móveis, o iminente surgimento da rede sem fio de terceira geração (3G) e a aparição da nova edição da plataforma Java para dispositivos móveis, tais como celulares, *paggers* e *palmtops*, vêm colaborar para o desenvolvimento de uma gama de aplicações ubíquas. Para as aplicações que requerem autonomia, mobilidade ou comportamentos sociais, aconselha-se o uso de técnicas provenientes da Inteligência Artificial, mas especificamente na área dos Agentes.

Neste trabalho de dissertação, objetivamos prover mecanismos para utilização de agentes nos dispositivos móveis. Para isto, tínhamos que prover dois componentes básicos: um para a comunicação entre os agentes de uma sociedade e outro para raciocínio embutido. Como os dois mecanismos deveriam executar (pelo menos alguma parte) dentro dos dispositivos, tínhamos que utilizar uma linguagem de programação apropriada e que fosse consenso entre boa parte da indústria de dispositivos. J2ME foi a linguagem escolhida, uma vez que foi desenvolvida especialmente para tais dispositivos e além disso é a única linguagem fruto de um consórcio entre os maiores representantes da indústria de telecomunicações (e.g. *Motorola*, *Siemens* e *Nokia*) e a *Sun*. Além dessa vantagem que já é por si só gigantesca, J2ME é uma linguagem baseada em Java e por isso igualmente comprometida com a portabilidade das aplicações, característica importante neste domínio.

Para o caso da comunicação entre os agentes, utilizamos como base para as implementações um sistema desenvolvido na Universidade de São Paulo (USP) chamada SACI, que provê uma infra-estrutura para comunicação entre agentes situados em uma sociedade. O nosso trabalho foi estender a arquitetura SACI para suportar os agentes residentes em dispositivos móveis. A dificuldade maior para a implementação desse componente foi contornar a ausência de RMI (*Remote Method Invocation*) usado pela arquitetura SACI. Para isso, provemos uma implementação que utiliza HTTP para a comunicação dos agentes e a batizamos de KSACI, o primeiro, juntamente com LEAP [12], sistema implementado em J2ME a prover um mecanismo de comunicação entre agentes.

Para o componente de raciocínio, deveríamos desenvolver um mecanismo para o raciocínio *embutido*, uma vez que não fazia muito sentido utilizar uma arquitetura cliente-servidor nesse contexto onde as conexões nem sempre estão disponíveis ou são muito caras. Para se embutir um mecanismo de inferência, existem várias possibilidades que foram identificadas e analisadas. Depois desta análise, escolhemos a mais coerente dentro do nosso contexto e implementamos o KEOPS baseado na arquitetura do sistema de produção com objetos embutidos, o JEOPS, desenvolvido na Universidade Federal de Pernambuco (UFPE). Para o desenvolvimento do KEOPS, encontramos problemas referentes a utilização de mecanismos de reflexão Java em JEOPS não suportados em J2ME que foram contornados com a modificação da arquitetura JEOPS. O KEOPS é, até onde sabemos, a primeira ferramenta que provê raciocínio embutido em dispositivos móveis.

Os dois sistemas foram testados individualmente em condições ideais. Separadamente tiveram performances aceitáveis. Sobre as limitações e problemas ainda não resolvidos no desenvolvimento dos dois sistemas pode-se citar com relação ao KSACI, a definição do esquema de identificação de classes codificadas em XML pelo sistemas SACI. Essa limitação impossibilita a decodificação de mensagens codificadas em XML pelos agentes não implementados com o KSACI. Agentes KSACI não sofrem, pois sua API suporta o decodificação de conteúdos XML codificados com o próprio KSACI.

Quanto ao KEOPS, observamos o crescimento quase exponencial das classes geradas pelo compilador JEOPS. Bases com muitas regras tendem a ter classes que as representam muito grandes. Por exemplo, uma base com 6 regras e 3 declarações em média por regra, gera classes com 54 KB. Uma base com 10 regras e também com 3 declarações por regra, tem o tamanho de 256 KB, o que ainda é inaceitável para a grande maioria dos dispositivos atuais.

Em suma, as contribuições deste trabalho são os dois sistemas supracitados (*KSACI* e *KEOPS*) cuja combinação foi batizada de *kAgent*, um ambiente básico e inédito para o desenvolvimento de agentes em dispositivos móveis que abre caminho para o desenvolvimento de uma série de aplicações ubíquas.

Como perspectivas de trabalhos futuros temos a intenção de fazer testes do *KSACI* em ambientes reais, pois devido a ausência de infraestrutura de telecomunicações, não pôde ser realizada a tempo. Além disso, é nossa intenção fazer a adaptação do *KSACI* para suportar não só *KQML*, mas *FIPA* como linguagem de comunicação e avaliar as possibilidades de se usar novas tecnologias para comunicação de sistemas *Peer-to-Peer* (*JXTA* [95]) e para transporte de pacotes (*SOAP* [96]).

Outra opção interessante para o *KSACI* seria o estudo dos sistemas para trocas de mensagens instantâneas. Hoje o sistema de código aberto *Jabber* [XXX] dispõe de uma arquitetura bastante flexível e segura para trocas de mensagens instantâneas, inclusive baseadas em XML. A arquitetura *Jabber* é baseada numa plataforma cliente-servidora, isto é, existe um sistema servidor que registra os usuários e redireciona as mensagens transmitidas, e uma série de sistemas cliente que implementam uma interface comum que o servidor entende. Recenemente foi produzido um sistema cliente escrito em *J2ME*. Um estudo mais aprofundado de como se utilizar tanto o servidor *Jabber* como o cliente *Jabber* escrito em *J2ME* para prover as funcionalidades disponíveis na arquitetura *KSACI*, seria um trabalho bastante interessante.

Com relação ao *KEOPS*, um trabalho interessante seria estudar o esquema de compilação de regras do *JEOPS* para tentar minimizar o

excesso de código produzido. Não é um problema especificamente do KEOPS, mas sim do JEOPS, mas mesmo assim um trabalho interessante quando se trabalha com ambientes com restrições de espaço.

APÊNDICE A: ATOS DE FALA

As teorias lingüísticas estudam e pesquisam linguagens: como elas trabalham, como elas mudam, e como nós as usamos. Em particular, a Teoria dos Atos da Fala criada por *Austin* (1962) [60], e posteriormente desenvolvida pelo filósofo *John Searle* [60], estuda as relações entre as expressões vocais e o desempenho, os efeitos e as mudanças no ambiente, produzidas por sentenças pronunciadas em linguagem natural.

Os atos de fala são elementos numa estrutura de conversa, que definem os possíveis cursos de eventos que podem ocorrer numa conversa entre duas pessoas [60].

A teoria de atos de fala está primeiramente relacionada com as expressões vocais da linguagem natural. *Austin* notificou que as sentenças expressadas pelos humanos durante a comunicação nem sempre afirmam um fato, mas na verdade executam algumas ações e que expressões vocais realizam ações com "força" diferenciada.

Para explicar melhor a teoria de atos de fala suponha que alguém diga: "Eu prometo lhe dar dez dólares", o que é uma promessa. Esta promessa é formada por vocábulos da fala humana. Se este alguém não usar estas palavras, ou palavras equivalentes, há uma grande possibilidade de não haver nenhuma promessa. Isso é a essência do

ato de fala; o pronunciamento das palavras gera a ação. Neste exemplo, a escolha de palavras (em particular, a palavra *promessa*) define o tipo de ato de fala executado. Mas isso nem sempre é o caso. Considere a sentença:

"Há um touro no campo"

Dependendo do contexto, a ação gerada pela expressão pode ser bastante diferente, senão veja:

- Se o emissor está descrevendo uma fazenda, a sentença significaria uma afirmação ou declaração.
- Se o emissor disser a alguém que está indo em direção ao campo, a sentença significará uma advertência.
- Se o emissor pronunciar a sentença quando alguém próximo está querendo "cruzar" sua vaca, então ela significará um pequeno conselho.

O tipo de ato de fala executado através de palavras particulares freqüentemente depende da intenção do emissor e o contexto no qual as palavras são proferidas.

Estes exemplos ilustram que palavras pronunciadas nem sempre definem exclusivamente o tipo de ato de fala executado pronunciando estas palavras. Por esta razão, Searle classificou os atos de fala segundo 3 aspectos:

- *Ato Locucionário*: emissão de palavras e sentenças com algum significado;
- *Efeito Ilocucionário*: corresponde a intenção do emissor com a mensagem emitida;
- *Efeito Perlocucionário*: corresponde o real efeito da mensagem no receptor.

O mesmo ato locucionário pode ter diferentes efeitos ilocucionários, dependendo do contexto no qual aquele ato é executado. Por exemplo, "Feche a porta!", é um ato locucionário com

um efeito ilocucionário de ordem para que o receptor fecha a porta. A reação do receptor ao ato locucionário é dita um efeito perlocucionário, qualquer que seja ele. É importante notar que o efeito ilocucionário nem sempre é o mesmo do efeito perlocucionário para um determinado ato locucionário, ou seja, nem toda mensagem é compreendida pelo receptor; pode haver incompreensão entre o que foi dito e o que foi realmente entendido.

9.1. TAXONOMIA

O conjunto de atos de fala reconhecido na maioria das comunidades lingüísticas é muito maior e muito mais especificado que a taxonomia proposta inicialmente por Austin. Searle (1972) fez várias críticas à taxonomia de Austin e propôs uma outra com as categorias descritas na abaixo.

Ato da Fala	Efeito Ilocucionário
<i>Assertives</i>	Representam um estado do mundo (afirmação, reivindicação, descrição, etc) <i>Ex.: "-A porta está fechada"</i>
<i>Commissives</i>	Confinam o emissor a alguma ação futura (promessas, ameaças, votos, etc) <i>Ex.: "-Fecharei a porta"</i>
<i>Directives</i>	Fazer com que o receptor execute alguma ação, (comando, pedido, desafio, solicitação, etc). <i>Ex.: "-Feche a porta!"</i>
<i>Declarations</i>	Provocar mudanças no estado do mundo (casar, nomear, abençoar, prender, etc) <i>Ex.: "-Nomeio este portão de Golden Gate"</i>
<i>Expressives</i>	Indicam o estado ou atitude mental emissor (cumprimentar, felicitar, agradecer, se desculpar, etc). <i>Ex.: "-Muito obrigado pelo presente"</i>

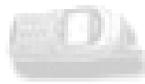
Verdicatives	Avaliam ou passam julgamento (julgar, tolerar, permitir)
	<i>Ex: "-Eu não tolero mais você!"</i>

Tabela 0.1: Categorias dos Atos de Fala [60]

Expressar uma primitiva de ato de fala com sucesso (ato perlocucionário) significa mudar o ambiente por meio das palavras pronunciadas. Por exemplo: "eu os declaro marido e mulher" representa um ato de sucesso quando o emissor (padre) e os ouvintes (noivos) estão nas condições de executar aquela ação (neste caso os noivos irão se comportar como marido e mulher desde então).

As condições de êxito de primitivas de expressões vocais são importantes para especificar como e quando elas são válidas numa real situação. Uma primitiva de expressão vocal pode ser nula ou sem êxito, de duas formas:

- As condições nas quais a expressão vocal é executada não satisfazem as exigências para que a expressão tenha sucesso ("Eu batizo pingüins");
- A expressão vocal é emitida sem sinceridade, como por exemplo, o emissor não está na posição para pronunciar uma certa oração. "Eu te batizo" sem estar numa posição que lhe permita fazer isso, ou seja, o emissor não é um padre.



10.

REFERÊNCIAS

- [01] Feng, Y.; Zhu, J. *Wireless Java Programming with Java 2 Micro Edition*, Indianapolis: Sams Publishing, ISBN: 0672321351, 2001.
- [02] WAP Specification. [Online] Disponível em: www.wapforum.org/what/technical.htm. Acessado em 22/maio/2002.
- [03] NTT DoCoMo iMode. [Online] Disponível em: www.nttdocomo.co.jp/english/imode. Acessado em 22/maio/2002.
- [04] Huang, A. ; Ling, B. ; Ponnekanti, S.; Fox, A. Pervasive Computing: What Is It Good For?. In: *Workshop on Mobile Data Management (MobiDE) in conjunction with ACM MobiCom '99*. Seattle, 1999.
- [05] Java2 Platform, Micro Edition (J2ME). [Online] Disponível em: java.sun.com/j2me. Acesso em 22/maio/2002.
- [06] Cunha, P. Projeto para o Instituto de Computação Ubíqua - Cubiq. Projeto Interno do Centro de Informática, Universidade Federal de Pernambuco, Recife, 2001.
- [07] Jennings, N.; Sycara, K.; Woolridge, M. A Roadmap of Agent Research and Development. In: *Journal of Autonomous Agents and Multi-Agent Systems*. Boston: Kluiver Academic Press, 1998, pp. 275-306.
- [08] Figueira, C. S.; Ramalho, G. Jeops - the Java Embedded Object Production System. In M. Monard e J. Sichman (eds.). *Advances in Artificial Intelligence*. London: Springer-Verlag Lecture Notes on Artificial Intelligence, 2000, v.1952, pp 52-61.

- [09] Albuquerque, R. ; Hübner, J. ; Paula, G. ; et all. KSACI: A Handheld Device Infrastructure for Agent Communication. In: *Proceedings of 8th International Workshop on Agents Theories, Architectures and Languages (ATAL'01)*. Seattle: Spring-Verlag, 2001.
- [10] Hübner, J.; Sichman, J. SACI: Uma Ferramenta para Implementação e Monitoração da Comunicação entre Agentes. In: *Proceedings International Joint Conference, 15th Brazilian Artificial Intelligence Symposium and 7th Ibero-American Artificial Intelligence Conference*, Atibaia, 2000, pp. 47-56.
- [11] Finin, T.; Labrou, Y.; Mayfield, J. KQML as an Agent Communication Language, In: *Software Agents*, J.M. Bradshaw (ed.). Cambridge: MIT Press, 1997, pp. 248-260.
- [12] Bergenti, F.; Poggi, A. LEAP: A FIPA Platform for Handheld and Mobile Devices. In: *Proceedings of 8th International Workshop on Agents Theories, Architectures and Languages (ATAL'01)*. Seattle: Spring-Verlag, 2001.
- [13] Laukkanen, M.; Tarkoma, S; Leinonen, J. FIPA-OS Agent Platform for Small-footprint Devices. In: *Proceedings of 8th International Workshop on Agents Theories, Architectures and Languages (ATAL'01)*. Seattle: Spring-Verlag, 2001.
- [14] Framingham, MA. Worldwide Mobile and Wireless Professional ServicesMarket will Increase 58.5% to 30 Billion in 2006, IDC Research Reveals. [Online] Disponível em: www.idc.com/getdoc.jhtml?containerId=pr2002_06_27_113728. Acessado em 10/agosto/2002.
- [15] Russel, S.; Norvig, P. *Artificial Intelligence: a Modern Approach*, Upper Saddle River: Prentice Hall, ISBN: 0131038052, 1994.
- [16] Mitchell, T. M. *Machine Learning*, Carnegie Mellon University, Boston: McGraw Hill College Div, ISBN: 0070428077, 1997.
- [17] UML - Unified Modeling Language. [Online] Disponível em: <http://www.omg.org/technology/documents/formal/uml.htm>. Acessado em 22/fevereiro/2003.
- [18] SQL

- [19] Riley, G. What Are Expert Systems?. [Online] Disponível em: www.ghgcorp.com/clips/ExpertSystems.html. Acessado em 22/maio/2002.
- [20] The W3C XML Extensible Markup Language Working Group Homepage. [Online] Disponível em: www.w3.org/XML. Acessado em 22/maio/2002.
- [21] RDF. [Online] Disponível em: www.w3.org/RDF. Acessado em 22/maio/2002.
- [22] Genesereth, M.; Fikes, R. (eds.) *Knowledge Interchange Format, Version 3.0 Reference Manual*. Technical Report Logic-92-1, Computer Science Department, Stanford University, 1992.
- [23] Bratko, I. *PROLOG Programming for Artificial Intelligence*, 3.ed., New York: Addison-Wesley Pub Co., ISBN: 0201403757, 2000.
- [24] OWL - Web Ontology Language. [Online] Disponível em: <http://www.w3.org/TR/owl-ref/>. Acessado em 22/fevereiro/2003.
- [25] The RuleML Markup Initiative. [Online] Disponível em: <http://www.dfki.uni-kl.de/ruleml/>. Acessado em 22/fevereiro/2003.
- [26] Wei, Fang. *F-Logic Semantics and Implementation of Internet Metadata*. Institut fuer Informatik, Computational Logic Universitaet Freiburg, 1999.
- [27] TCP/IP. [Online] Disponível em: webopedia.internet.com/TERM/T/TCP_IP.html. Acessado em 12/outubro/2002.
- [28] DCOM. [Online] Disponível em: webopedia.internet.com/TERM/D/DCOM.html. Acessado em 12/outubro/2002.
- [29] CORBA. [Online] Disponível em: webopedia.internet.com/TERM/C/CORBA.html. Acessado em 12/outubro/2002.
- [30] RMI. [Online] Disponível em: java.sun.com/products/jdk/rmi. Acessado em 12/outubro/2002.
- [31] SOAP - Simple Object Access Protocol. [Online] Disponível em: <http://www.w3.org/TR/SOAP/>. Acessado em 22/fevereiro/2003.

- [32] Gruber, T. What is an Ontology? [Online] Disponível em: www-ksl.stanford.edu/kst/what-is-an-ontology.html. Acessado em 22/maio/2002.
- [33] QUALCOMM BREW Homepage. [Online] Disponível em: www.qualcomm.com/cda/brew. Acessado em 22/maio/2002.
- [34] QUALCOMM Homepage. [Online] Disponível em: www.qualcomm.com. Acessado em 22/maio/2002.
- [35] QUALCOMM BREW Whitepaper. [Online] Disponível em: www.qualcomm.com/cda/brew2/0,1817,92,00.html. Acessado em 22/maio/2002.
- [36] Yuri, F. Que tal um Java de bolso? Revista Info Exame, São Paulo (SP), v.186, Zona Wireless, 2001.
- [37] Java 2 Platform, Standard Edition (J2SE). [Online] Disponível em: java.sun.com/j2se. Acesso em 22/maio/2002.
- [38] Java Card. [Online] Disponível em: java.sun.com/products/javacard. Acessado em 22/maio/2002.
- [39] Embedded Java. [Online] Disponível em: java.sun.com/products/embeddedjava. Acessado em 22/maio/2002.
- [40] Personal Java. [Online] Disponível em: java.sun.com/products/personaljava. Acessado em 22/maio/2002.
- [41] Java 2 Platform, Enterprise Edition (J2EE). [Online] Disponível em: java.sun.com/j2ee. Acesso em 22/maio/2002.
- [42] JavaOne 1999. [Online] Disponível em: java.sun.com/javaone/javaone99. Acessado em 22/maio/2002.
- [43] CDC and the CVM Virtual Machine. [Online] Disponível em: java.sun.com/products/cdc. Acessado em 22/maio/2002.
- [44] CLDC and the K Virtual Machine (KVM). [Online] Disponível em: java.sun.com/products/cldc. Acessado em 22/maio/2002.
- [45] White, J. JavaPro, Big Plans for J2ME. [Online] Disponível em: www.devx.com/premier/mgznarch/javapro/2001/05may01/jw0105/jw0105p.asp. Acessado em 22/maio/2002.

- [46] Mobile Information Device Profile (MIDP). [Online] Disponível em: java.sun.com/products/midp. Acesso em 22/maio/2002.
- [47] Foundation Profile. [Online] Disponível em: java.sun.com/products/foundation. Acessado em 22/maio/2002.
- [48] PDA Profile for the J2ME Platform. Java Community Process. Disponível em: jcp.org/jsr/detail/075.jsp. Acessado em 22/maio/2002.
- [49] Java 2 Platform Micro Edition, Wireless Toolkit. [Online] Disponível em java.sun.com/products/j2mewtoolkit/. Acessado em 22/maio/2002.
- [50] MIDP for Palm OS, [Online] Disponível em: java.sun.com/products/midp4palm/index.html. Acessado em 22/maio/2002
- [51] Java Remote Method Invocation (RMI). [Online] Disponível em: java.sun.com/products/jdk/rmi. Acessado em 22/maio/2002.
- [52] Giguere, E. Connected Device Configuration and the Foundation Profile. [Online] Disponível em: wireless.java.sun.com/foundation/ttips/cdcfoundation. Acessado em 22/maio/2002.
- [53] Personal Profile Specification (JSR 62). [Online] Disponível em: jcp.org/jsr/detail/062.jsp. Acessado em 22/maio/2002.
- [54] Day, B. Developing Wireless Applications using the Java 2 Platform, Micro Edition. [Online] Disponível em: wireless.java.sun.com/getstart/articles/wirelessdev/wirelessdev.pdf. Acessado em 22/maio/2002.
- [55] JavaMobiles. [Online] Disponível em: www.javamobiles.com. Acessado em 22/maio/2002.
- [56] Murray, C. Cell phone makers jump for Java. EETimes. [Online] Disponível em: www.eetimes.com/printableArticle?doc_id=OEG20010607S0102. Acessado em 22/maio/2002.
- [57] Mayfield, J., Labrou Y. et al. Evaluation of KQML as an Agent Communication Language. In: *Intelligent Agents II: Agent Theories, Architectures and Languages*. Montreal: Springer-Verlag Lecture Notes in Artificial Intelligence, 1996, v.1037, pp. 347-360.
- [58] DARPA Knowledge Sharing Effort. [Online] Disponível em: www.cs.umbc.edu/kse. Acessado em 22/maio/2002.

- [59] Ontolingua. [Online] Disponível em: www.ksl.stanford.edu/software/ontolingua/. Acessado em 22/maio/2002.
- [60] Tavares, J. M. S. B. Comunicação Entre Agentes Inteligentes. Trabalho Final de Graduação, 1999.
- [61] FIPA - The Foundation for Intelligent Physical Agents. [Online] Disponível em: www.fipa.org. Acessado em 22/maio/2002.
- [62] CGI. [Online] Disponível em: isp.webopedia.com/TERM/C/CGI.html. Acessado em 12/outubro/2002.
- [63] PERL. [Online] Disponível em: www.perl.com. Acessado em 12/outubro/2002.
- [64] PHP: Hipertext Processor. [Online] Disponível em: www.php.net. Acessado em 12/outubro/2002.
- [65] ASP. [Online] Disponível em: www.asp.net. Acessado em 12/outubro/2002.
- [66] Java Servlet Technology. [Online] Disponível em: java.sun.com/products/servlet. Acessado em 12/outubro/2002.
- [67] JSP - Java Server Pages Specification. [Online] Disponível em: <http://jcp.org/aboutJava/communityprocess/first/jsr152/index2.html>. Acessado em 22/fevereiro/2003.
- [68] W3C - World Wide Web Consortium. [Online] Disponível em: <http://www.w3.org/>. Acessado em 22/fevereiro/2003.
- [69] DAML - The DARPA Agent Markup Language Homepage. [Online] Disponível em: <http://www.daml.org/>. Acessado em 22/fevereiro/2003.
- [70] DAML-OIL. [Online] Disponível em: <http://www.daml.org/2001/03/daml+oil-index>. Acessado em 22/fevereiro/2003.
- [71] Shoham, Y. Agent-Oriented Programming. In: *Artificial Intelligence*, 1993, v.60, pp. 51-92.
- [72] Fisher, M. A survey of Concurrent METATEM - the language and its applications. In: *Temporal Logic - Proceedings of the First International Conference*. Berlim: Springer-Verlag Lecture Notes in Artificial Intelligence, 1994, v.827, pp. 480-505.

- [73] Thomas, S. The PLACA Agent Programming Language. In: *Intelligent Agents II: Agent Theories, Architectures and Languages*. Amsterdam: Springer-Verlag Lecture Notes in Artificial Intelligence, 1995, v. 890, pp. 355-369.
- [74] Masini, G.; Napoli, A.; Colnet, D. *Object-Oriented Languages*. London, San Diego, New York, Boston, Sydney, Tokyo: Academic Press, ISBN: 0124773907, 1992.
- [75] Ait-Kaci, H. et al. *The Wild LIFE Handbook*. Paris: Digital Research Laboratory, 1994.
- [76] JACK Intelligent Agents. [Online] Disponível em: www.agent-software.com/shared/home. Acessado em 22/maio/2002.
- [77] KIEV Language Home Page. [Online] Disponível em: www.forestro.com/kyev. Acessado em 22/maio/2002.
- [78] Almgren, J. et al. Mixing Java and Prolog. [Online] Disponível em: cswww.essex.ac.uk/TechnicalGroup/sicstus/sicstus_12.html. Acessado em 22/maio/2002.
- [79] Calejo, M. Java plus Prolog systems. [Online] Disponível em: dev.servisoft.pt/interprolog/systems.htm. Acessado em 22/maio/2002.
- [80] Minerva. [Online] Disponível em: www.ifcomputer.com/MINERVA. Acessado em 22/maio/2002.
- [81] jProlog. [Online] Disponível em: www.cs.kuleuven.ac.be/~bmd/PrologInJava. Acessado em 22/maio/2002.
- [82] Prolog Cafe. [Online] Disponível em: kaminari.scitec.kobe-u.ac.jp/PrologCafe. Acessado em 22/maio/2002.
- [83] Jinni. [Online] Disponível em: www.binnetcorp.com/Jinni. Acessado em 22/maio/2002.
- [84] Forgy, C. RAL/C and RAL/C++: Rule-Based Extensions to C and C++. In: *Proceedings of the OOPSLA'94 workshop on Embedded Object-Oriented Production Systems (EOOPS)*. Paris: Laforia, 1994.
- [85] Pachet, F. Représentation de connaissances par objets et règles: le système NéOpus. Tese de Doutorado, Université Paris VI. Paris: Laforia. 1992.

- [86] Friedman-Hill, E. JESS, the Java Expert System Shell. [Online] Disponível em: herzberg.ca.sandia.gov/jess. Acessado em 22/maio/2002.
- [87] OPSJ: Rules for Java. [Online] Disponível em: www.pst.com/opsj.htm. Acessado em 22/maio/2002.
- [88] Figueira Filho, C. JEOPS - Integração de Entre Objetos e Regras de Produção em Java. Dissertação de Mestrado. Centro de Informática, Universidade Federal de Pernambuco. 2000.
- [89] Pachet, F. On the Embeddability of Production Rules in Object-Oriented Languages. In: *Journal of Object-Oriented Programming*, 1995, v.8, n.4, pp. 19-24.
- [90] Lindholm, T.; Yellin, F. The Java™ Virtual Machine Specification. Addison-Wesley, ISBN: 0201432943, 1999, 2.ed. Disponível em: <ftp://ftp.javasoft.com/docs/specs/vmspec.2nded.html.zip>. Acessado em 22/maio/2002.
- [91] IBM: VisualAge for Java. 2000. [Online] Disponível em: www.software.ibm.com/vajava. Acessado em 22/maio/2002.
- [92] Java Reflection. [Online] Disponível em: java.sun.com/j2se/1.3/docs/guide/reflection. Acessado em 22/maio/2002.
- [93] Friedman-Hill, E. The JESS Mailing List. [Online] Disponível em: herzberg.ca.sandia.gov/jess/mailling_list.html. Acessado em 22/maio/2002.
- [94] Sun Microsystems. Java Compile and Runtime Environments. 2000. [Online] Disponível em: java.sun.com/docs/white/platform/javaplatform.doc3.html. Acessado em 22/maio/2002.
- [95] Project JXTA. [Online] Disponível em: jxme.jxta.org. Acessado em 22/maio/2002.
- [96] kSOAP Project. [Online] Disponível em: ksoap.enhydra.org. Acessado em 22/maio/2002.
- [97] Java Card. [Online] Disponível em: java.sun.com/products/javacard. Acessado em 12/outubro/2002.