Arquitectura de Agentes:
Desenho e Fundações de um Modelo de Negócio

por

José Palmeiro

Orientado por

Professor Doutor Paulo Quaresma

Mestrado em Engenharia Informática
Departamento de Informática
Universidade de Évora
Outubro 2006

Tese submetida em conformidade com os requisitos
para obtenção do grau de Mestre em Engenharia Informática

Arquitectura de Agentes:
Desenho e Fundações de um Modelo de Negócio

por

José Palmeiro

Orientado por

Professor Doutor Paulo Quaresma

Mestrado em Engenharia Informática
Departamento de Informática
Universidade de Évora
Outubro 2006

Tese submetida em conformidade com os requisitos
para obtenção do grau de Mestre em Engenharia Informática

Sumário

A presente dissertação descreve um estudo arquitectural na área de agentes, *internet* e serviços. O estudo, contextualizado numa base funcional para recuperação de informação, introduz novas aproximações na modelação de um sistema de negócio.

A solução proposta é baseada numa arquitectura de agentes, exposta em serviços e disponibilizada numa aplicação *web*. Este sistema multi-agente apresenta-se como alternativa aos modelos clássicos, comuns em plataformas empresariais, encapsulando toda a lógica de negócio com elevados padrões de integração e reutilização.

Do ponto de vista arquitectural é estudado todo o desenho assim como várias perspectivas de integração. É proposto um canal de comunicação com os agentes via aplicação *web* e apresentada uma forma de expor parte do sistema numa *Service Oriented Architecture*. São ainda exploradas tecnologias assíncronas para comunicação com a plataforma de agentes, através de *Asynchronous Javascript And XML*. Foi também alvo de estudo a utilização de *Dynamic Logic Programming* em agentes que interagem com o utilizador.

Esta dissertação apresenta portanto uma forma de analisar, desenhar e implementar uma arquitectura centrada em agentes, segundo as melhores práticas da engenharia de *software*. Também estabelece técnicas de modelação com as tecnologias usadas, de forma isolada ou composta.

Abstract

Agents Architecture:
Design and Foundations of a Business Model

This thesis describes an architectural study on agents, internet and services. The study is presented in an information retrieval context, and introduces new approaches on business logic modeling.

The proposed solution is based on an agents architecture, powering a web application and exposed in services. This multi-agent system stands as an alternative approach to enterprise modeling, holding all business logic with high integration patterns and reuse.

Several integration perspectives are also studied from an architectural point of view. It is presented a way of exposing this system in a Service Oriented Architecture, and proposed a middleware communication channel between agents and the web layer. Asynchronous Javascript And XML technologies are explored in communications with the agents platform, and the use of Dynamic Logic Programming in user agents is also a target study.

The analysis, design and implementation phases of an agent-centric platform are presented in this thesis, following software engineering best practices. It also defines new modeling practices with the used technologies, in an isolated or compound fashion.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Objectivos

O objectivo desta dissertação é apresentar um estudo arquitectural na área de agentes, *internet* e serviços. Por ser uma tese no ramo de engenharia, o estudo é baseado no desenvolvimento de um engenho: uma arquitectura de *software* - contextualizada numa base funcional orientada para a recuperação de informação.

O desenvolvimento prático apresenta como objectivo principal o estudo de uma plataforma de agentes, exposta numa arquitectura orientada a serviços e disponibilizada numa aplicação *web*. A solução proposta é centrada numa arquitectura de agentes e apresenta-se como alternativa aos modelos clássicos usado nas plataformas empresariais, sendo potencialmente integrável e reutilizável para vários fins. Dada a natureza de um sistema multi-agente, é também objectivo estudar um canal assíncrono de comunicação com a aplicação *web*.

O estudo apresenta ainda o objectivo de explorar o paradigma *Dynamic Logic Programming* (DLP) na modelação interna de alguns agentes, assim como a sua transformação em *web services*.

Através do desenho detalhado que compõe esta dissertação, é possível descobrir e inferir caminhos a seguir, e detectar passos a evitar ou contornar, com a utilização destas tecnologias de forma conjunta, ou isolada. Espera-se ainda que esta tese ajude a reduzir o espaço que separa as tecnologias usadas na indústria, das tecnologias de investigação teórica e de Inteligência Artificial (IA).

1.2 Organização do Documento

A presente dissertação divide-se em três grandes partes. Como objectivo desta divisão está a facilidade e objectividade da leitura, podendo o leitor navegar e seleccionar com maior facilidade os diferentes tópicos. É ainda objectivo fornecer um documento com princípio meio e fim, acessível a todo o tipo de leitores.

A primeira parte apresenta um enquadramento teórico, de leitura opcional, uma vez compreendidos os conceitos usados no restante da dissertação. A segunda parte foca-se na solução estudada e desenvolvida no âmbito da tese, uma Arquitectura de Agentes apresentada desde a visão do sistema até ao seu desenho detalhado. É uma parte bastante técnica, e orientada para a descrição e compreensão das partes mais importantes do sistema. A terceira e última é dedicada à análise do software desenvolvido.

A primeira parte divide-se em três capítulos. O primeiro capítulo introduz os conceitos mais relevantes sobre a tecnologia de agentes, apresentando ao leitor uma perspectiva histórica, passando pela definição e análise dos diferentes tipos de agentes, finalizando

com uma descrição do sistema multi-agente. O segundo capítulo apresenta uma breve descrição das tecnologias *Dynamic Logic Programming* (DLP). O terceiro capítulo fecha a componente teórica e introduz de forma muito prática as arquitecturas orientadas a serviços. Todas estas tecnologias estão presentes e são usadas na segunda parte da tese.

A segunda parte está dividida em quatro capítulos, orientados pelo *Rational Unified Process* (RUP). O primeiro introduz o modelo funcional do *software* desenvolvido, através dos clássicos diagramas de casos de uso, posteriormente refinados em diagramas de actividade, sendo o início do processo de desenvolvimento de *software*. O segundo capítulo descreve toda a arquitectura da solução proposta, através da apresentação das diversas vistas que se destinam a modelar os diferentes aspectos da arquitectura de *software* da aplicação desenvolvida. Neste capítulo, o leitor é introduzido aos diversos diagramas de sequência e colaboração de fluxos arquitecturalmente significativos, acompanhados de todo o desenho detalhado, bibliotecas, e padrões de desenho usados. O terceiro capítulo apresenta a arquitectura de *software* para o componente JLups, um componente DLP usado na solução. Tal como o capítulo anterior, este também descreve as diversas vistas arquitecturais de forma adaptada a um componente de *software*. O quarto e último capítulo desta parte, resume todos os outros com menos detalhe técnico, sendo orientado para leitores tecnicamente mais leigos. Embora seja um resumo, pode ser um ponto de partida para uma leitura mais profunda, pois fornece ao leitor um visão de alto nível sobre toda a componente técnica.

Como pré-requisitos para leitura desta parte, o leitor deverá estar familiarizado com a *Unified Modeling Language* (UML).

A terceira e última parte está dividida em dois capítulos. O primeiro capítulo é dedicado à componente de testes ao *software* desenvolvido. São apresentados resultados de vários fluxos significativos, em diversas camadas e módulos. O último capítulo é dedicado às conclusões, apresentando diversas notas arquitecturais sobre todos os pontos tecnicamente importantes da presente dissertação. É concluído com uma análise do trabalho futuro a elaborar às diversas camadas de *software*.

1.3 Contribuições

De acordo com os objectivos considerados foi possível estabelecer os seguintes avanços técnicos:

- Apresentar a análise, desenho e implementação de uma plataforma centrada em agentes, exposta numa *Service Oriented Architecture* (SOA) e integrada numa aplicação *web*;
- Desenvolver um sistema multi-agente desenhado segundo as melhores práticas - laborais orientadas à engenharia de *software* - sendo usado para encapsular a lógica de negócio da plataforma;
- Descrever e propor uma abordagem aos agentes numa SOA, sem necessidade de redesenhar as fundações;
- Explorar tecnicamente o uso de padrões de desenho bem conhecidos na definição da plataforma, promovendo a correcta e essencial separação dos diferentes estilos e camadas arquitecturais - Agentes, Serviços e *Web*;
- Apresentar vários exemplos de interacção *web* de forma assíncrona via *Asynchronous Javascript And XML* (AJAX), implementada com ferramentas actuais, que

demonstram como este tipo de tecnologia pode ser utilizado em projectos Java EE e agentes;

- Propor uma utilização concreta das tecnologias DLP em agentes que interagem com o utilizador e residentes num ambiente empresarial;
- Desenvolver o *middleware* AgentServices para comunicação com agentes via plataforma *web*;
- Desenvolver o componente JLups, uma versão Java da linguagem LUPS.

Parte I

Enquadramento Teórico

Capítulo 2

Agentes

A tecnologia de agentes é um tema bastante discutido ao longo do tempo. A palavra "agente" foi e ainda é, alvo de muito uso e abuso. Ainda assim é um conceito importante e tecnologicamente interessante. É com base nesta visão que este capítulo foi escrito, realizando-se uma perspectiva geral sobre a tecnologia em causa. Foi considerado importante focar as definições e características de agentes, tal como as vertentes distribuídas.

2.1 Perspectiva Histórica

Os agentes inteligentes têm origem em investigações realizadas na área de robótica e de inteligência artificial, em meados dos anos 70. Comummente, considera-se o projecto ELIZA[1] como o primeiro *software* inteligente desenvolvido, como forma de "computador terapeuta" que conseguia manter um diálogo com um utilizador [MJ98].

À parte do projecto ELIZA, pouca atenção foi dada ao *software* inteligente até à criação da *World Wide Web* (WWW), no início dos anos 90. Com o elevado crescimento ao nível de conteúdos, foi necessário construir o primeiro motor de busca. O *World Wide Web Worm*, assim chamado, foi usado para descobrir e contabilizar o número de servidores na *web*. Com o passar do tempo, agentes apelidados de *crawlers*, ou *spiders*, com o objectivo de pesquisarem novas páginas e indexá-las, tornaram-se os primeiros agentes inteligentes a serem usados em grande escala.

No final dos anos 90, agentes mais recentes e mais evoluídos foram desenvolvidos para o comércio electrónico. Em 1997 foi criado o agente *RoboChopper*, um dos primeiros destinado a auxiliar o utilizador a efectuar compras na *web*, através da localização de itens e comparação de preços [MJ98, p.14]. A Microsoft, nesse mesmo ano, também criou um agente interactivo de nome *Office Assistant* com o objectivo de auxiliar o utilizador a operar com um produto. A eficiência deste agente é discutível. O seu objectivo não.

Actualmente, a inter-operabilidade entre agentes é uma realidade. Vários agentes interagem uns com os outros com um determinado objectivo. Os *standards* de comunicação estão construídos e bem definidos. No entanto a tecnologia de agentes sobrevive embora com pouca aceitação no mercado de trabalho. As plataformas continuam a evoluir e a apresentar novos argumentar de forma a tentar conquistar o seu espaço num mercado competitivo.

[1] Ver http://www.manifestation.com/neurotoys/eliza.php3

2.2 Agentes Inteligentes

"*a self-contained, interactive and concurrently-executing object, with some encapsulated internal state and which could respond to messages from other similar objects*" Carl Hewitt, 1997

2.2.1 Enquadramento

Em meados dos anos 50, J. McCarthy e G. Selfridge, desenvolveram as primeiras ideias sobre o actual conceito de agentes. Propuseram um sistema flexível que, quando fornecido um objectivo, realizasse as operações necessárias para o atingir e sempre que ficasse bloqueado, poderia perguntar e receber conselhos em termos humanos [Ehl01, p.15]. O sistema seria um "*soft robot*" que viveria e realizaria o seu trabalho dentro de um computador. Nos anos 70, Carl Hewitt refinou esta ideia e propôs um objecto "*self-contained, interactive and concurrently-executing*", que denominou de "actor". Pela sua definição, um actor era um agente computacional que dispunha de um endereço de correio e um determinado comportamento. Estes actores comunicavam através de *message-passing* e concretizavam as suas acções de forma concorrente. Dispunham também de um estado interno e podiam responder a outros agentes.

A tecnologia de agentes, tal como os objectos de programação numa linguagem evoluída, têm origem na área de programação distribuída e inteligência artificial [SV97]. As suas semelhanças e origem são bastante próximas. Ambas as tecnologias tentam resolver problemas em determinadas situações, embora a orientação por agentes vá um passo à frente, apresentando um objectivo claro na sua existência. A área de *Distributed Artificial Intelligence* (DAI), foca-se no estudo da distribuição e coordenação de conhecimento e acções em ambientes com múltiplas entidades [Ehl01, p.15]. Pode ainda ser dividida em dois três subconjuntos:

- *Distributed Problem Solving* (DPS), soluciona problemas através da divisão de trabalho por módulos ou nós, que por sua vez cooperam e partilham conhecimento;
- *Multi-Agent Systems* (MAS), representa o estudo da coordenação de um grupo de agentes inteligentes autónomos;
- *Parallel AI* (PAI), usa a computação paralela para melhorar o desempenho nas áreas de DAI.

No final dos anos 90, os Agentes e os MAS emergiram como um subconjunto da DAI. Um sistema de agentes é um MAS [Les95] ou uma agência [AR96]. Em contraste com o DPS, a investigação em MAS[2] está concentrada no comportamento de uma colecção de agentes autónomos que colaboram para atingir um objectivo.

Agentes inteligentes são programas de *software* com diferentes tipos de características, existindo principalmente com o objectivo de auxiliar o seu utilizador. Humano ou não. Por exemplo, pode programar-se um agente para coleccionar informação da *web* sobre um determinado tema. A ideia principal é poupar tempo ao utilizador delegando todo o tipo de processos a um agente.

Os agentes podem ser vistos como um passo natural para um novo paradigma de *software*, onde a tecnologia de agentes se constrói sobre tecnologias anteriores [AR99, p.18]. Cada objecto na tecnologia de agentes encapsula o seu próprio código, dados, invocação, estado e objectivo. Bradshaw [Bra97, p.28] apresenta uma interessante definição de agente:

[2]Tema desenvolvido na secção 2.3 Sistemas Multi-Agente.

"*Agent-oriented programming can be thought of as a specialization of object-oriented programming approach, with constraints on what kinds of state-defining parameters, message types, and methods are appropriate. From this perspective, an agent is essentially 'an object with an attitude'.*"

Nwana [Nwa96] divide o estudo de agentes em duas fases: a primeira em cerca de 1977, e a segunda por volta de 1990. A primeira fase, cujas raízes estão principalmente na DAI, concentrou-se fortemente em agentes deliberativos com modelos simbólicos internos. Este trabalho contribuiu para a melhor compreensão na colaboração de agentes, na decomposição e distribuição de tarefas, resolução de conflitos via negociação, entre outros. A última fase é mais recente e estuda um tipos de agentes mais abrangente. A ênfase mudou de deliberação para actuação, de raciocínio para acções remotas [Bra97, p.4].

A noção de agentes como uma aproximação à programação não é uma realidade actual. Embora o seu futuro não esteja certo, os agentes inteligentes recebem actualmente um crescente atenção. O futuro desta tecnologia pode passar por três alternativas [AR99, p.19]: aumento da utilização e amadurecimento da tecnologia; estagnação e transformação da tecnologia num paradigma de programação a par com os objectos; tornar-se uma tecnologia restrita usada por poucos investigadores.

A investigação actual foca-se muito em agentes deliberativos, que usam modelos de raciocínio simbólico, interacção, cooperação e coordenação entre agentes. A descoberta de vários tipos de agentes é recente e data a partir de 1990, com o auxílio da *internet* e com os avanços na área de Inteligência Artificial (IA). É ainda importante salientar que para uma tecnologia deste tipo ter sucesso, são necessários *standards*. Estes *standards* existem e serão abordados nos pontos seguintes.

2.2.2 Definições

Agente apresenta o significado de "que age", "que actua", "que cuida de negócios alheios" ou "pessoa encarregada de praticar certas operações materiais por outrem". Algumas empresas e vários investigadores têm-se dedicado à área de agentes, tendo por sua vez resultado em diversas definições. Por não existir um consenso serão apresentadas várias definições, que ilustram as dificuldades existentes na construção de uma definição de agente universalmente aceite.

2.2.2.1 O Agente de Nwana

Nwana [Nwa96] admite a dificuldade da definição. Refere ainda que é fácil cair-se na definição de um componente de *software* capaz de agir de forma a realizar tarefas para proveito do utilizador. A palavra agente por ser usada para vários fins e em várias áreas, gera alguns conflitos. De forma a descrever o termo agente, Nwana define vários tipos, classificando-os de acordo com os atributos que exibem:

- Autonomia;
- Aprendizagem;
- Cooperação.

De forma a ser um agente inteligente, este deve apresentar pelo menos duas destas qualidades.

Um agente é autónomo se pode operar por si só sem necessidade de intervenção humana. Dispõe de um objectivo e estado, e age de forma a concretizar o seu objectivo.

A pro-actividade, ou a capacidade de iniciativa própria é uma característica muito importante. Através da cooperação com outros agentes, podem-se concretizar tarefas de maior complexidade. Para os agentes serem inteligentes ou "espertos", devem apresentar a capacidade de aprendizagem. Este processo desenvolve-se durante a interacção ou reacção com o ambiente externo.

Com estes três atributos mínimos, podem identificar-se quatro tipos de agentes:

- *Collaborative Agents*;
- *Collaborative Learning Agents*;
- *Interface Agents*;
- *Smart Agents.*

Nada fora da área de intersecção é considerado agente. Preferencialmente um agente deverá possuir três atributos embora este caso esteja um pouco longe da realidade.

Nwana [Nwa96] refere ainda que existem duas outras dimensões, na qual um agente pode ser inserido. A primeira diz respeita à mobilidade, e considera se um agente é móvel ou estático. Um agente móvel pode circular por uma rede, enquanto um estático não tem esta habilidade. A outra dimensão classifica os agentes de acordo com o seu perfil, podendo ser deliberativos ou reactivos. O primeiros derivam do paradigma do pensamento deliberativo, possuindo um modelo interno de raciocínio onde abordam e negoceiam com outros agentes de forma a atingir o seu objectivo. O agentes reactivos actuam através de um comportamento estímulo/resposta. Estas duas novas dimensões não invalidam a necessidade de um agente apresentar duas das três características acima referidas.

Note-se que Nwana omite a expressão "inteligente" no seu discurso.

2.2.2.2 O Agente de Foner

De acordo com Foner [Fon97, p.35], um agente deve preencher um determinado número de requisitos:

- Autonomia: Um agente deve apresentar um certo grau de autonomia do seu utilizador. Esta autonomia requer acções periódicas, execuções espontâneas e iniciativa, em que o agente será capaz de executar acções com preempção ou independentes para benefício do utilizador;
- Personalidade: O objectivo de um agente é facilitar a execução de uma tarefa a um utilizador. Como os utilizadores não realizam todos as mesmas tarefas (e mesmo os que as realizam não o fazem da mesma forma) um agente deve ser educado para um determinado tipo de tarefa. Idealmente, devem existir componentes de aprendizagem, de forma a evitar que o utilizador programe explicitamente o agente, e de memória, para que a aprendizagem não se repita vezes sem fim;
- Discurso[3]: É necessário que o agente partilhe a agenda do utilizador e execute as tarefas da forma pretendida por este. Normalmente esta situação requer uma forma de comunicação[4] bidireccional entre agente e utilizador. Esta comunicação pode

[3] *Discourse*, no original.

[4] Por esta métrica, por exemplo, o *Garbage Collector* (GC) da linguagem Java não é um agente. Embora o GC trabalhe com o objectivo de libertar e gerir recursos de memória, não existe nenhuma forma de comunicação entre o utilizador/programador e este componente. Por outro lado, também não existem nenhuma forma de memória entre utilizador e GC.

apresentar a forma de uma "conversa" simples até um discurso de alto nível em que o utilizador e agente interagem repetidamente, e ambos recordam interacções anteriores;

- Risco e Confiança: A ideia de agente está intimamente relacionada com a noção de delegação. Não se pode delegar uma tarefa a uma entidade que não garanta ou não compreenda minimamente o problema e solução em causa, de acordo com as especificações. No entanto, por definição delegar implica renunciar ao controlo de uma operação, remetendo-o para outra entidade, com diferente memória, experiências ou agenda. Existe então um risco de o agente errar e confiança de poder acertar. Um balanço portanto. Uma decisão deve ponderar o grau de risco e ponderar domínio de confiança, isto é, os custos de um possível erro, normalmente associado a uma análise de risco;

- Domínio: O domínio de interesse é crucial. Se é um jogo ou passatempo, o risco é normalmente baixo e pode investir-se no agente com um elevado grau de confiança. Por outro lado, se é uma transacção bancária o custo é elevado, e muita das vezes é proibido falhar;

- *Graceful Degradation*: Os agentes funcionam melhor quando apresentam *graceful degradation* em casos de comunicações falhadas, ou em domínios perdidos. É preferível que algumas das tarefas possam ser concluídas com sucesso, do que nenhuma o seja. Esta situação gera confiança entre agente e utilizador;

- Cooperação: O agente e o utilizador cooperam para atingir um determinado objectivo. O utilizador especifica as acções a serem tomadas, enquanto o agente descreve o que pode fazer e devolve resultados. Neste tipo de protocolo (comunicação bidireccional), cada entidade pode consultar a outra de forma a verificar o estado actual da interacção. As duas entidades interagem como *peers* em sistemas orientados por agentes[5].

- Antropomorfismo: Característica que torna os agentes semelhantes a humanos. Um agente pode ou não apresentar esta característica, embora esta esteja fortemente relacionada com agentes. É importante salientar que esta característica não transforma um programa num agente;

- Expectativa: A interacção entre utilizador e agentes tem maior sucesso se o agente se comportar da forma desejada. É importante que as expectativas do utilizador perante o agente se tornem realidade.

De acordo com Foner [Fon97], o programa Julia é um agente que preenche a totalidade dos seus requisitos. Julia é uma[6] participante de um jogo numa *Multi User Dimension* (MUD), tal como qualquer outro jogador. Este agente armazena informação sobre o estado do jogo. Um jogador pode ganhar vantagem se aceder a Julia e conseguir extrair-lhe informação. A Julia mantém informação sobre as salas no jogos e sobre outros jogadores, lembrando-se de situações concretas.

2.2.2.3 O Agente de Petrie

Petrie [Pet96, p.2] critica severamente o agente de Foner, Julia (ver secção 2.2.2.2), referindo que esta não se distingue da maior parte de *software* que corre em background

[5]Em sistemas não orientado por agentes, o utilizador comanda uma acção tipicamente através de uma interface e provavelmente nunca lhe é submetido uma pergunta a não ser que algo corra mal.

[6]Julia em [Fon97, p.1] afirma ser fêmea.

e responde a perguntas. Refere ainda que esta não demonstra iniciativa, nem interrompe jogadores a não ser para entregar mensagens. Apenas fala quando é abordada. Não a considera portanto um agente inteligente. O significado da palavra inteligente no contexto de agente é considerado problemático por Petrie [Pet96, pp.1-2], referindo três problemas quando se tenta definir um agente como inteligente:

1. O significado da palavra inteligente é subjectivo, isto é, depende da opinião do observador que interage com o agente em causa;

2. A expressão anterior é subjectiva e só se aplica a epifenómenos, em vez de a objectivos de desenho. À parte de passar um teste de Turing, ninguém desenvolve um agente inteligente sem objectivos. Um agente é desenvolvido de forma a concretizar uma tarefa. Criar agentes com o objectivo de serem inteligentes é um objectivo pobre;

3. Embora existindo várias definições de inteligência, o seu maior ónus é não distinguir de forma clara o *software* resultante de outras tecnologias que clamam inteligência como atributo. Diferentes pessoas usam diferentes significados.

Petrie incide muito na expressão "autónomo". Refere que em vez do termo inteligência, devia incidir-se mais na definição de autonomia, como forma de separação entre agentes e outras tecnologias. A introdução deste conceito como parte da definição de agentes, e o afastamento da inteligência como pré-requisito, resume de certa forma a intenção de Petrie.

2.2.2.4 O Agente de Jennings e Wooldridge

Jennings e Wooldridge [JW98, p.4] definem agente como um sistema computacional situado num determinado ambiente, capaz de realizar acções autónomas de forma a concretizar os seus objectivos. O conceito de autonomia surge no sentido de o agente ser capaz de concretizar as suas tarefas sem o auxilio de intervenção humana, ou de outros agentes, tendo controlo sobre as suas acções e estado interno. Um agente encapsula o seu comportamento. Comparando um agente a um objecto[7], que também mantém um estado interno, pode-se detectar uma diferença importante: existe pelo menos um método num objecto[8] que pode ser invocado por outro componente. Esta situação implica que um objecto não é autónomo. De acordo com Jennings e Wooldridge, com agentes não sucede o mesmo. Estes apresentam controlo sobre as suas acções, solicitando a outros agentes a realização de determinadas tarefas. Contrariamente aos objectos, não invocam métodos de outros agentes.

Existem vários exemplos de sistemas autónomos. Por exemplo, um simples termóstato que actua quando determinadas condições se estabelecem. Jennings e Wooldridge [JW98, p.4] definem este tipo de sistemas como agentes. Diferenciam ainda de forma veemente, agente de agente inteligente. O caso do termóstato, ou de um outro sistema agente mais complexo, não são considerados agentes inteligentes. Um agente inteligente é um sistema computacional que é capaz de acções autónomas flexíveis, de forma a concretizar os seus objectivos. A flexibilidade é introduzida com o significado de um sistema apresentar as seguintes características:

- Responsivo/Reactivo: Os agentes devem compreender o ambiente envolvente (que pode ser a *internet*, um grupo de agentes, um utilizador, o mundo físico, entre outros) e responder a alterações que sucedam nele;

[7]Referente a uma linguagem de programação orientada por objectos.
[8]Considerando que o objecto apresenta alguma utilidade.

- Pró-activo: Os agentes não devem somente agir em resposta ao seu ambiente, apresentando sentido de oportunidade, comportamento orientado por objectivos e iniciativa;

- Social: Os agentes devem conseguir interagir quando julguem apropriado, com outros agentes artificiais e humanos, de forma a solucionar o problema em causa e auxiliar outros nas suas actividades.

Jennings e Wooldridge acreditam que estes quatro atributos (autónomo, responsivo, pró-activo e social) diferenciam um agente inteligente de outras entidades. Estes quatro atributos não excluem a possibilidade de apresentarem atributos adicionais. Também tornam claro que a flexibilidade separa agentes de agentes inteligentes.

2.2.2.5 O Agente de Maes

De acordo com Maes [Mae95, p.108], agentes autónomos são sistemas computacionais que habitam num ambiente complexo, dinâmico, interpretando e agindo autonomamente nesse mesmo ambiente de forma a concretizar os seus objectivos. Estes agentes também podem tomar diferentes formas dependendo da natureza do ambiente em que habitam. Maes descreve [Mae95, p.111] várias características que um agente deve considerar, nomeadamente melhorar o seu desempenho com base em experiências passadas. Por outras palavras, aprendizagem. Deve também conseguir comunicar com outros agentes no seu mundo, humanos ou artificiais. Salienta ainda diversas características importantes como rapidez, reactividade, adaptabilidade, robustez, autonomia e "*lifelike*". Esta última característica refere-se a um comportamento não mecânico, não previsível e espontâneo.

2.2.2.6 O Agente de Hayes-Roth

Hayes-Roth [HR95, p.3] descreve um agente inteligente como algo que realiza três funções:

1. Percepção de condições dinâmicas no meio ambiente;

2. Acções que afectem condições nesse mesmo meio;

3. Raciocínio de forma a interpretar percepções, resolver problemas e determinar acções.

Resumindo, um agente primeiro detecta um evento no meio ambiente, raciocina sobre ele determinando as acções em causa, e finalmente actua sobre essa decisão.

A um nível conceptual, a percepção informa o raciocínio e o raciocínio guia as acções, embora por vezes percepções possam conduzir directamente a acções. Esta definição abstracta permite a existência de una grande variedade de agentes biológicos e artificiais, cujas capacidades variam desde extremamente limitadas, comportamentos estereotipados, até comportamentos extremamente sofisticados e versáteis [HR95, p.3].

De forma a compreender esta definição, é importante perceber que Hayes-Roth enfatiza a importância de definir um agente específico de acordo com o seu objectivo. Hayes-Roth [HR95, p.7] refere ainda que um agente deve apresentar um elevado grau de adaptabilidade, modificando o seu comportamento de acordo com a sua envolvente. Um agente pertence portanto a um domínio específico, e deve ser construído de forma a agir nesse mesmo domínio.

2.2.2.7 O Agente de Franklin e Graesser

Por Franklin e Graesser [FG96, p.4] cada agente:

- Está situado dentro e é parte integrante de um ambiente;
- Percepciona esse ambiente e actua de forma autónoma sobre ele;
- Não necessita de entidades que lhe forneçam *input* ou interpretem ou usem o *ouput*;
- Age de acordo com a sua agenda;
- Influencia o meio ambiente;
- Actua durante um período de tempo.

Estes requisitos constituem a essência de ser um agente. Transpondo-a para uma definição:

"An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future." [FG96, p.4]

Esta definição, embora geral, resume de forma clara os requisitos de ser agente. É ainda demasiado abrangente para ser útil, necessitando da adição de novos requisitos de forma a produzir subclasses de agentes [FG96, p.4].

Também distinguem de forma simples um programa de um agente. Um programa ordinário pode percepcionar a sua envolvente com o seu *input*, embora normalmente o seu *ouput* não afecte as suas posteriores percepções. Também falha o teste de continuidade, pois normalmente é executado uma vez, entrando em "*stand-by*" até à próxima chamada. A maior parte dos programas apresentam uma ou até duas destas qualidades, sem considerar a definição de ambiente. Todo os agentes são programas, embora nem todos os programas sejam agentes [FG96, p.5].

2.2.3 Propriedades de um Agente

Na secção anterior foram analisadas várias definições introduzidas por diversos investigadores. Nota-se que não existe um consenso, existindo n definições para n diferentes investigadores. Cada um define agente de acordo com a sua visão ou compreensão, resultando num conceito algo subjectivo. No entanto é também interessante notar que existem várias semelhanças por entre as diversas descrições. Alguns conceitos são usados da mesma forma com o objectivo de descrever um agente. Nesta secção resumem-se as propriedades principais associadas a um agente, sendo as primeiras cinco tipicamente consideradas mais importantes.

2.2.3.1 Autonomia

A autonomia está intimamente ligada à definição de agente. A maior parte das definições pondera esta propriedade, considerando-a como uma das principais e mais importantes.

Pela Porto Editora, autonomia é "autodeterminação; regulamentação dos próprios interesses; possibilidade que uma entidade tem de estabelecer as suas próprias normas; independência". No presente contexto pode ser considerada como a capacidade de realizar acções de forma independente, sem necessidade de acompanhamento do utilizador. O agente deve por si só determinar a melhor solução e agir por sua iniciativa. Embora seja uma propriedade muito citada, nem todos os estão de acordo com o seu significado. Por exemplo, Foner (ver 2.2.2.2) considera Julia um agente autónomo e com iniciativa,

enquanto Petrie (ver 2.2.2.3) pelo contrário não a acha diferente de um vulgar programa de *software* que responde a perguntas.

2.2.3.2 Adaptabilidade/Aprendizagem

Apesar de nem todos os agentes serem capazes de adaptar o seu comportamento, a propriedade de adaptação é usualmente considerada como um sinónimo de inteligência.

Um agente adapta-se através da aprendizagem. Tal como Maes (ver 2.2.2.5) refere, um agente melhora o seu desempenho com base em experiências passadas. Estas experiência desenvolvem-se durante a interacção ou reacção com o ambiente externo em que o agente se encontra. Embora seja uma característica aparentemente importante, muitos investigadores como o caso de Jennings e Wooldridge (ver 2.2.2.4) não a mencionam. Existem também muitas formas de aprendizagem, variando desde um simples armazenamento de informação até à imitação de acções realizadas por o utilizador.

2.2.3.3 Comunicação

Um agente deve ser capaz de comunicar com outros agentes ou humanos. Esta comunicação, por Ehlert [Ehl01, p.17], deve ser feita através de uma linguagem de comunicação expressiva, idealmente assemelhando-se a um discurso humano.

Destacam-se então dois tipos de comunicação, em que uma por ser entre agentes poderá ser estruturada através de linguagens implementadas para o efeito, enquanto a outra poderá usar simbolismos ou língua natural, para comunicação com humanos.

2.2.3.4 Cooperação/Social

Os agentes devem ser capazes de comunicar uns com outros, ou com humanos, de forma a cooperarem e realizarem tarefas complexas que doutra forma não conseguiriam sozinhos. Devem também ajudar outros agentes nas suas tarefas.

Através da cooperação, tarefas complexas podem ser divididas por agentes de forma a que cada um fique com tarefas mais simples, resultando numa diminuição da complexidade. Pode existir cooperação entre agentes com o objectivo de encontrarem em conjunto uma solução, ou para mera distribuição de tarefas.

Esta característica por motivos lógicos está intimamente ligada à anterior, apresentando-se ainda como uma das grandes vantagens em usar agentes.

2.2.3.5 Responsivo/Reactivo

Um agente deve conseguir compreender o seu meio ambiente e ser capaz de realizar acções baseadas nestas percepções.

2.2.3.6 Mobilidade

A capacidade de um agente se mover de um sistema para outro de forma a aceder a recursos remotos, ou a outros agentes. Este sistema pode ser uma rede, como por exemplo a *internet*.

Muitos investigadores acreditam que os agentes móveis irão oferecer um novo e importante método de *Information Retrieval* (IR) e de transacções em redes. Por exemplo, um agente pode navegar na *internet* em busca do voo mais barato para um certo destino. O agente viaja de servidor em servidor de forma a seleccionar a melhor informação, sem necessidade do utilizador estar *online* o tempo todo. Quando a melhor informação

for seleccionada, somente esta é enviada para a máquina origem sendo posteriormente apresentada ao utilizador.

2.2.3.7 Personalidade/*Lifelike*

O objectivo de inserir personalidade num agente é criar a ilusão de sentimentos, emoções e interacção social. A criação deste tipo personagens requer um variado tipo de tecnologia, incluindo reconhecimento de discursos, língua natural, animações e síntese de discurso [Bra97, p.26]. Bradshaw refere ainda que são essenciais os mecanismos de compreensão de diálogos e psicologia social.

Este tipo de características distinguiria certamente agentes de outro tipo de *software*, ao mesmo tempo que a interacção entre utilizador e agentes seria mais harmoniosa.

2.2.3.8 Proactividade

Os agentes não se devem restringir somente a respostas, mas devem também apresentar comportamento oportunista e orientado a objectivos, com iniciativa quando apropriado. É importante que os agentes se afastem da passividade e se tornem activos, apresentando sugestões ao utilizador.

2.2.3.9 Reprodução

Através da reprodução, os agentes podem evoluir e adaptar-se a alterações no meio ambiente, aumentando assim as hipóteses de sobrevivência. Por outras palavras, assistir-se-ia a uma evolução "artificial" em todo análoga à "natural". A reprodução é um dos tópicos principais na Vida Artificial[9].

2.2.3.10 Continuidade Temporal

Característica que representa a persistência de identidade e estado em largos períodos de tempo.

2.2.3.11 Capacidade de Inferência

Um agente pode agir na especificação de tarefas abstractas usando conhecimento prévio de objectivos gerais.

2.2.4 Tipos de Agentes

Com base nas características descritas em 2.2.2.1, Nwana identifica e descreve oito tipos de agentes:

- Agentes Colaboradores;
- Agentes Pessoais ou Interfaces;
- Agentes Móveis;
- Agentes da *internet* ou Informativos;

[9]Uma designação que nasceu pela mão de Christopher Langton. Literalmente, na definição dada pelo fundador, tratam-se de formas de vida fabricadas tecnologicamente pelo homem em vez de criadas pela Natureza. Estas criaturas de tipo sintético, digital ou mesmo físico apresentam a particularidade de se poder desenvolver autónoma e independentemente, de replicarem no seu comportamento algumas das leis essenciais da vida tal como a conhecemos, sem termos de as comandar a cada passo.

- Agentes Reactivos;
- Agentes Híbridos;
- Smart Agents;
- Sistemas de Agentes Heterogéneos.

2.2.4.1 Agentes Colaboradores

Autonomia e cooperação são características que predominam neste tipo de agente. Embora possível, a aprendizagem não é o seu foco principal. Tipicamente dispõem de capacidade social, poder de resposta e pro-actividade, sendo capazes de agir com racionalidade em ambientes multi-agentes e com constrangimentos temporais. A negociação com outros agentes também é parte integrante do processo de preparação de tarefas. São usualmente estáticos e *coarse-grained*, embora possam ser benévolos[10]. A maioria destes tipo de agentes não realiza uma aprendizagem complexa.

A motivação em sistemas de agentes colaboradores pode incluir uma das seguintes razões (que são especializações das motivações para DAI):

- Resolução de problemas demasiado complexos para um simples agente resolver, devido a limitação de recursos ou do risco dum sistema centralizado;
- Permitir a interligação e interoperabilidade de múltiplos sistemas, por exemplo sistemas de suporte à decisão, entre outros;
- Fornecer soluções para problemas distribuídos, por exemplo controlo de tráfego aéreo;
- Fornecer soluções onde o conhecimento está representado numa forma distribuída;
- Realçar a modularidade (reduzindo a complexidade), a rapidez (devido ao paralelismo), grau de confiança (devido à redundância), flexibilidade e reutilização ao nível de conhecimento (devido à partilha de recursos);
- Investigação de outros temas, como a interacção entre sociedades humanas.

Este tipo de agente apresenta-se muito atractivo para aplicações industriais, existindo alguns exemplos reais como o Projecto Pleiades[11].

2.2.4.2 Agentes Pessoais ou Interfaces

Um agente pessoal, como o nome indica, realiza tarefas para o seu proprietário ou utilizador. Para tal, apresenta como características principais autonomia e aprendizagem. Este tipo de agente pode ser visto como um assistente pessoal que colabora com o utilizador no mesmo ambiente de trabalho. É importante notar que existe uma diferença substancial entre a colaboração com o utilizador e colaboração com outros agentes. Esta última, normalmente necessita de uma linguagem de comunicação de forma a os agentes se compreenderem.

Talvez a forma mais simples de definir um agente deste tipo será compreendê-lo como um programa que usa técnicas de aprendizagem máquina, de forma a fornecer assistência

[10] No presente contexto apresenta um significado contrário a competição. Um agente pode competir com outros agentes.

[11] Ver http://www2.cs.cmu.edu/afs/cs.cmu.edu/project/theo5/www/pleiades.html.

a um utilizador numa determinada aplicação, por exemplo, uma folha de cálculo ou até uma navegação da *internet*.

Tipicamente, o agente observa e monitoriza as acções tomadas pelo utilizador através do interface do mesmo, aprendendo e sugerindo quando possível. Ao nível da aprendizagem, estes agentes usam normalmente quatro técnicas [Mae94, pp. 33-34]:

1. O agente aprende continuamente através das acções do utilizador;

2. Outra forma de aprendizagem é através de *feedback* directo e indirecto do utilizador. O *feedback* indirecto ocorre quando o utilizador rejeita uma sugestão do agente, tomando uma outra acção. O utilizador pode ainda fornecer explicitamente *feedback* negativo a acções automatizadas pelo agente;

3. O agente pode aprender a partir de exemplos fornecidos pelo utilizador. Este pode treinar o agente fornecendo-lhe exemplos hipotéticos de eventos e situações, e demonstrando-lhe o que fazer nesses casos;

4. A última forma de aprendizagem é pedir conselhos a outros agentes que assistiram outros utilizadores em tarefas semelhantes, portanto com um nível de experiência superior. Um agente pode também adquirir graus de confiança superiores em determinados agentes que provaram exactidão no passado.

Este tipo de agente consegue reduzir e automatizar tarefas para o utilizador e aumentar a sua eficiência no decorrer do tempo, visto usufruir de um forte poder de adaptabilidade às preferências e hábitos do utilizador.

A principal critica é a tendência a funcionar com unidade *stand-alone*, comunicando pouco e apenas com semelhantes. A comunicação com outro tipo de agente poderia fornecer uma mais valia para as suas capacidades.

2.2.4.3 Agentes Móveis

O agentes móveis constituem uma ferramenta de grande interesse para a maioria de aplicações que exigem um elevado grau de automatização. São processos computacionais capazes de percorrer vastas áreas, tais como a WWW. Interagem com hospedeiros remotos, armazenado informação para o utilizador e retornando ao local da partida aquando da satisfação dos seus objectivos - definidos pelo utilizador.

No entanto, a mobilidade não é uma característica necessária ou suficiente para se ser agente. O agentes móveis são agentes porque apresentam autonomia e cooperam, embora de forma diferente dos agentes colaboradores (ver 2.2.4.1). Por exemplo, podem cooperar ou comunicar através do fornecimento da localização de objectos internos ou métodos a outros agentes. Através desta técnica, um agente pode trocar apenas os dados ou informação necessária com outros agentes.

Os benefícios destes agentes não são maioritariamente funcionais, pois as tarefas de um agente móvel podem igualmente ser feitas por um agente estático. Imagine-se a situação em que um utilizador escrevesse um programa que reservasse um bilhete de avião, para uma determinada localização, num determinada horário e a um determinado preço. Se o agente for estático, terá que consultar todas as bases de dados das agências em busca de um voo compatível, efectuando várias consultas e recuperando muita informação, ocupando por sua vez muita largura de banda. O tempo para a realização da tarefa poderia variar bastante.

Por outro lado, se o agente apresentar mobilidade poderá "viajar" de umas agências para outras, realizando consultas locais e armazenando apenas informação importante,

voltando por fim ao local de origem com uma suposta melhor opção. Aqui o utilizador pode aceitar ou recusar a proposta do agente.

As vantagens deste tipo de agente em relação aos seus homólogos são:

- Custos reduzidos de comunicação, como pode ser verificado no exemplo acima;
- Recursos locais reduzidos, que torna atraente a utilização de agentes em dispositivos de recursos limitados (por exemplo, telemóveis, *smart-phones*, *Personal Digital Assistant* (PDA), entre outros);
- Fácil coordenação, pois é mais fácil coordenar um número de pedidos remotos e independentes e somente tratar os resultados localmente;
- Computação Assíncrona, podendo desligar-se o dispositivo ou realizar outro tipo de tarefas;
- Fornece um ambiente natural de desenvolvimento para implementar serviços comerciais;
- Uma arquitectura distribuída flexível;
- Fornece um novo paradigma para desenho de processos.

A segurança é normalmente apontada como o maior problema dos agentes móveis, impedindo a sua disseminação como paradigma.

Estes problemas podem ser caracterizados da seguinte maneira:

- Agentes maliciosos, é importante proteger os servidores contra agentes maliciosos. Estes tipo de programas atacam geralmente por exploração não autorizada de acessos, *denial-of-service*, e forjamento de identidade;
- Hospedeiros maliciosos, os agentes devem estar protegidos de hospedeiros maliciosos, contra diversos tipos de manipulação, monitorização, informação forjada ou perigosa.

Aquando o desenvolvimento de aplicações com agentes móveis, podem ser considerados dois cenários [Mar03, p.76].

O primeiro passa por lançar a aplicação num ambiente fechado. Neste caso é possível identificar uma autoridade central controladora dos nós da rede. Esta autoridade é também responsável por atribuir permissões aos utilizadores para a criação de agentes. Existe um ambiente controlado. Ao contrário do que se possa pensar, este tipo de cenário pode ser muito favorável para vários tipos de aplicações, nomeadamente gestão de rede, aplicações de telecomunicações, entre outros. Neste tipo de ambiente, as soluções comuns de segurança (autenticação, mecanismos de autorização associados a algoritmos de chave assimétricas) são suficientes.

O segundo cenário surge quando a aplicação corre em ambiente aberto. Nesta situação os agentes migram para diferentes nós controlados por diferentes autoridades, possivelmente com diferentes objectivos. Um exemplo clássico duma situação deste tipo é uma aplicação de comércio electrónico na *internet*. Muitos *sites* podem disponibilizar uma plataforma de agentes, permitindo a migração destes para os seus nós, onde se realizam consultas sobre produtos e preços, e transacções. Neste caso os *sites* irão competir uns com os outros para que os agentes realizem as transacções neles próprios. Não existe uma autoridade central, sendo possível cada *site* atacar agentes, roubar-lhes informação e obrigá-los a realizarem operações que não era suposto fazerem. São ainda necessárias algumas evoluções para usar aplicações deste tipo em ambientes abertos.

[Mar03, p.77] refere ainda que a justificação para a falta de segurança em aplicações com agentes móveis é falsa. Podem actualmente desenvolver-se aplicações seguras. Por outro lado, a falta de segurança não impediu o desenvolvimento de várias outras tecnologias.

2.2.4.4 Agentes da *Internet* ou Informativos

Os agentes informativos devem a sua origem às largas quantidades de informação actualmente disponíveis. Foram criados para lidarem com tarefas hoje impossíveis para humanos, o tratamento e gestão de grandes quantidades de informação.

Diferenciam-se dos agentes colaboradores (ver 2.2.4.1) por serem definidos pelo que fazem, e não pelo que são. Existem ainda duas razões principais para se desenvolverem agentes informativos. A primeira e já referida necessidade de lidar com grandes quantidades de informação, e a segunda motivos económicos.

Este tipo de agente pode apresentar mobilidade, atravessando a *internet* recolhendo informação para devolver à origem. São usualmente vistos como estáticos, usando motores de busca ou mecanismos de *caching* para recuperar informação considerada importante. Existem semelhanças claras entre estes agentes e agentes móveis e interfaces. O futuro destes agentes pode passar certamente por agentes móveis (ver 2.2.4.3), ou se forem estáticos, por interfaces (ver 2.2.4.2). As críticas a estes agentes podem retirar-se dos pontos 2.2.4.3 e 2.2.4.2, caso sejam móveis ou estáticos, respectivamente.

2.2.4.5 Agentes Reactivos

Também conhecidos por agentes reflexivos, reagem com regras estímulo-resposta ao estado do meio ambiente. Representam uma categoria especial de agentes que não possuí modelos simbólicos internos do meio em que se encontram, apresentando total incapacidade de previsão de acontecimentos.

Como vantagem principal apresentam robustez e rapidez de resposta. O facto de não apresentarem memória é uma séria limitação. A sua simplicidade e funcionalidade de comportamento está directamente associada à capacidade criativa e intelectual do seu criador. Podem apresentar padrões de comportamento muito complexos, através da interacção de múltiplos agentes reactivos.

Os agentes deliberativos e reactivos são dois extremos. Muitos agentes podem ser caracterizados por apresentarem qualidades de ambos.

2.2.4.6 Agentes Híbridos

Dos cinco tipos de agentes anteriormente estudados, todos apresentam prós e contras. De forma a diminuir as deficiências e maximizar as qualidades, foram criados os agentes híbridos. Este agentes são uma combinação de dois ou mais tipo de agentes.

O objectivo é conseguirem atingir um nível de eficiência superior à conseguida por um tipo de agente simples. Por exemplo, unir qualidades do paradigma deliberativo e reactivo, criaria um agente robusto, com rápido poder de resposta e adaptável, onde a componente deliberativa orientaria o agente para determinados objectivos de longo prazo, podendo eventualmente colaborar com outros agentes.

2.2.4.7 Smart Agents

Ver secção 2.2.2.1.

2.2.4.8 Sistema de Agentes Heterogéneos

Um sistema heterogéneo de agentes, ao contrário dum híbrido, refere-se a um sistema constituído com pelo menos dois tipos diferentes de agentes. Pode também conter agentes híbridos.

A ideia principal que está por detrás deste tipo de sistema, é a interacção entre diferentes agentes, onde cada um está especializado em determinadas tarefas. Este domínio conhecido por *agent-based software engineering*, foi criado para facilitar a inter-operabilidade entre agentes. O ponto chave para esta inter-operabilidade entre agentes heterogéneos reside na linguagem de comunicação de agentes - *Agent Communication Language* (ACL).

2.3 Sistemas Multi-Agente

Numa análise simples ao quotidiano laboral do ser humano, é fácil perceber que um grupo de pessoas é normalmente mais produtivo que uma só. Através do trabalho de equipa, conseguem-se desempenhos nunca outrora possíveis com somente um elemento, por melhor que ele fosse. Uma equipa de projecto na área das Tecnologias de Informação (TI), é normalmente constituída por diversos tipos de elementos, uns analistas, outros arquitectos, *developers*, *testers*, entre outros. Estas diferenças de conhecimento e especializações são essenciais e tornam uma equipa produtiva. Este pensamento transposto para a tecnologia de agentes pode ser definido como um sistema multi-agente. A definição de "sinergia" explica bem o conceito, sendo o efeito resultante da acção de diferentes agentes que actuam de modo semelhante, e de valor superior ao do conjunto desses agentes se actuassem de forma isolada. A soma das partes é maior que o todo. A diversidade de conhecimento e especializações são um ponto central neste tipo de sistemas.

Os *Multi-Agent Systems* (MAS) são caracterizados pela interacção entre dois ou mais agentes, com o intuito de realizar determinadas tarefas. Para tal colaboram e coordenam as suas acções. Esta área, embora seja parte integrante da DAI é largamente influenciada por muitos outros campos, como robótica, psicologia, biologia, vida artificial, entre outros.

Estes sistemas surgem principalmente da necessidade em construir sistemas com "peças" heterogéneas de *hardware* ou *software*, que interagissem umas com as outras sem necessidade de coordenação centralizada. O conceito de adaptabilidade e robustez foi a razão da remoção de sistemas centralizados. Por exemplo, um robot não poderia ir para um planeta longínquo pois a comunicação entre este e humanos não seria suficientemente rápida.

Devido à inspiração retirada de ciências sociais para a construção de sistemas multi-agentes, tal como da economia (cada agente dispõe de uma função para maximizar), psicologia baseada num modelo orientado a *Belief, Desires and Intentions* (BDI) (crenças, desejo, intenção, com noções de compromisso a tarefas e a semelhantes), sociologia baseada em noções especificas de organizações, tornou-se possível a realidade multi-agente. De acordo com a teoria da evolução, a complexidade de um ambiente conduz à adaptação das espécies. Neste caso, agentes.

2.3.1 Características

Por Vlassis [Vla03, pp.1-3] os aspectos que caracterizam um sistema multi-agente e os que os diferenciam de um sistema mono-agente podem dividir-se nas seguintes dimensões.

2.3.1.1 Desenho

Normalmente os agentes que pertencem a um MAS são desenhados de forma diferente. Um exemplo típico são os agentes de *software* (normalmente chamados de *softbots*) normalmente implementados por diferentes criadores. Geralmente as diferenças de desenho são ao nível do *hardware* (como por exemplo, os robots que jogam futebol), ou do *software* (agentes desenhados para sistemas de operação diferentes). Normalmente este tipo de agentes são chamados heterogéneos, em contraste com os homogéneos que são desenhados de forma idêntica e apresentam as mesmas capacidades. No entanto esta distinção não é muito clara, pois agentes que usam o mesmo *hardware* e *software* e apresentam um comportamento diferente são também chamados de heterogéneos.

É importante reter que a heterogeneidade pode influenciar todos os aspectos funcionais de um agente, desde a percepção até à tomada de decisões, enquanto num sistema com um único agente esta situação não acontece.

2.3.1.2 Ambiente

Os agentes podem navegar em ambientes dinâmicos ou estáticos. A maior parte das técnicas de IA para sistemas com um só agente foram desenvolvidas para ambientes estáticos, pois são mais fáceis de manipular e permitem um maior rigor. Por outro lado, num MAS a mera presença de múltiplos agentes torna o ambiente aparentemente dinâmico para cada um. Esta situação pode ser problemática em casos de aprendizagem, onde se podem detectar comportamentos pouco estáveis.

2.3.1.3 Percepção

A informação recuperada dos sensores de agentes num MAS é normalmente distribuída: os agentes podem observar dados que diferem no espaço (diferentes localizações), no tempo (chega a diferentes momentos) e na semântica (requer diferentes interpretações). Estas situações tornam o ambiente observável, que por sua vez irá influenciar as suas decisões.

2.3.1.4 Controlo

De forma oposta aos sistemas com um só agente, o controlo nos MAS é distribuído. Por outras palavras, descentralizado, o que significa que não existe nenhum processo central que armazene informação de cada agente e coordene as suas acções. O poder de decisão está então do lado de cada agente. O problema que existe nas decisões em MAS é um assunto respeitante à teoria de jogos. Com equipas cooperativas (por exemplo, uma equipa de futebol de robots) a tomada de decisões resulta assíncrona, o que dificulta a coordenação entre agentes. Uma boa coordenação implica uma boa decisão para o grupo.

2.3.1.5 Conhecimento

Assume-se que num sistema com um único agente, este seja consciente das suas acções mas não necessariamente como afecta o meio ambiente. Num MAS, o nível de conhecimento sobre o meio envolvente difere por cada agente. Por exemplo, numa equipa com dois agentes homogéneos cada agente pode saber o conjunto de possíveis acções do outro, podem ambos partilhar percepções, ou inferirem decisões a partir de uma base de conhecimento comum. Por outro lado, um agente que observe a equipa contrária normalmente não conhecerá o seu conjunto de acções nem percepções. Geralmente num

MAS, cada agente deve também considerar a informação de outros agentes para tomar decisões. Esta base de conhecimento comum é um conceito crucial nesta tecnologia.

2.3.1.6 Comunicação

A interacção está normalmente associada com alguma forma de comunicação. Tipicamente esta comunicação apresenta dois sentidos, onde qualquer agente pode ser emissor ou receptor. A comunicação pode ser usada em vários casos, como por exemplo negociação, coordenação, entre outros. A comunicação também levanta alguns problemas sobre quais os protocolos a serem usados para troca de informação, e qual a linguagem usada para a comunicação entre agentes, especialmente em ambiente heterogéneos. Pode ser descrita a nível mais formal ao considerar-se que cada primitiva de comunicação actualiza o conhecimento, e consequentemente o estado do agente. As primitivas de comunicação trocadas entre agentes são normalmente conhecidas por actos comunicativos. De entre os tipos mais comuns de comunicação podem destacar-se:

- *Informing*, Informações sobre o estado actual;
- *Querying*, Perguntas sobre estados;
- *Commiting*, Proceder a uma determinada acção;
- *Prohibiting*, Proibir uma acção;
- *Directing*, Indicar a um agente uma acção.

Cada acto de comunicação afecta um agente de forma diferente. Por exemplo, um acto informativo pode reduzir a incerteza de um agente sobre o seu estado e sobre o de outros (pode imaginar-se o caso de um jogo de futebol entre agentes, em que o possuidor da bola avisa aos restantes elementos da sua equipa que tem a bola).

Por outro lado, um acto de *commiting* pode ser usado para indicar um percurso a ser tomado pelo agente (ainda no contexto do exemplo anterior, um agente pode indicar que se vai mover para a esquerda no campo de jogo).

A coordenação é uma propriedade muito importante dum sistema de agentes. O grau de coordenação está relacionado com as interacções do sistema, com a necessidade de evitar actividades estranhas, evitar *deadlocks* e manter a aplicação num estado estável. Como Huhns e Stephens referem [Wei99, p.82], normalmente para uma cooperação ser bem sucedida um agente deve manter um modelo de outros agentes, e também desenvolver um outro para futuras interacções. Esta situação implica sociabilidade.

Existem linguagens e protocolos para comunicação entre agentes. As duas mais notáveis são a *Knowledge Query Manipulation Language* (KQML) e *Foundation for Intelligent Physical Agents* (FIPA) ACL, onde cada uma usa a sua própria sintaxe para os actos de comunicação.

2.3.2 Motivação, Aplicabilidade e Desafios

Tal como um simples agente, é difícil prever um conjunto de aplicações onde um MAS possa ser usado. Um sistema complexo de *software* pode ser abordado como um conjunto de pequenos agentes autónomos, onde cada um apresenta as suas funcionalidades e propriedades.

A motivações sobre o aumento do interesse sobre os MASs devem-se, segundo Sycara [Syc98, p.80] a:

- Resolver problemas demasiado complexos para um agente centralizado;

- Permitir a inter-conexão a inter-operabilidade por múltiplos sistemas;
- Fornecer soluções a problemas que podem naturalmente ser considerados como uma sociedade interactiva de agentes autónomos (por exemplo, agentes de negócios que podem comprar ou vender);
- Fornecer soluções que usem frequentemente fontes de informação distribuídas;
- Fornecer soluções onde se exija conhecimento distribuído;
- Melhorar o desempenho em várias dimensões, tais como:
  - Rapidez e eficiência, devido à computação assíncrona e paralela;
  - Robustez e segurança, onde um sistema exibe *graceful degradation* (ver 2.2.2.2) quando um ou mais agentes falham;
  - Escalabilidade e flexibilidade, visto ser fácil adicionar agentes ao sistema;
  - Custo, assumindo que o agente é uma unidade de baixo custo em relação ao sistema;
  - Desenvolvimento e reutilização, visto ser mais fácil desenvolver e manter *software* modular do que monolítico.

Um grande desafio actual é a *internet*. Nela existe uma grande variedade de agentes que navegam sem um protocolo e linguagens bem definidas. Um protocolo de comunicação ao nível de agentes está num nível superior ao *Transmission Control Protocol* (TCP). Num ambiente desta natureza, e como já referido em secções anteriores, a tecnologia MAS pode ser usada em prol do utilizador. Por exemplo para controlo de tráfego, ciências sociais, robótica, realidade virtual, jogos, entre outros.

2.3.3 Modelo BDI

Nesta secção apresenta-se uma arquitectura com os conceitos de crenças, desejos e intenções. Em inglês, uma arquitectura *Belief, Desires and Intentions* (BDI).

As arquitecturas BDI têm origem no trabalho do projecto da Rational Agency, em *Stanford Research Institute*, na década de oitenta. A origem deste modelo surge dos estudos da compreensão do raciocínio humano desenvolvido pelo Filósofo Micheal Bratman [Bra87], que foca muito do seu trabalho nesta área no papel das intenções no raciocínio pragmático[12]. As suas origens estão portanto na tradição filosófica para perceber o raciocínio, o processo de decidir, momento a momento qual a acção a tomar de acordo com os objectivos.

Os sistemas e formalismos que centram as atenções em intenções são normalmente referenciados como arquitecturas BDI [RG91, p.1]. Enquanto a maioria das teorias filosóficas reduzem as intenções a crenças e desejos, Bratman argumenta que as intenções apresentam um objectivo distinto nas raciocínio lógico, tratando-as como planos parciais para acções que o agente está disposto a seguir para atingir o seu objectivo.

2.3.3.1 Conceitos

Crença Pode ser interpretada como uma visão, um conceito, uma relação que o agente mantém sobre o mundo à sua volta. O agente usa as crenças para poder deliberar sobre o estados futuros. O conjunto das crenças pode de certa forma caracterizar

[12]Tradução de *Pratical Reasoning*, utilizações de raciocínio deverão ser consideradas neste sentido.

o estado actual de um sistema ou agente. Um agente pode ainda manter todo o tipo de crenças sobre o mundo, desde crenças sobre outros agentes, crenças sobre pessoas, até mesmo crenças sobre si mesmo. Uma crença é sobretudo um conhecimento;

Desejos Representam os estados desejáveis que o sistema poderá apresentar. Para um desejo se concretizar é necessário um conhecimento inerente a ele e um contexto favorável a sua realização. Os desejos motivam um agente a agir de forma a concretizar determinados objectivos. Podendo também ser interpretados como objectivos;

Intenções São um subconjunto de desejos. Se um agente decidir seguir um determinado caminho, então esse caminho torna-se uma intenção. As intenções determinam o processo de raciocínio pois são elas que determinam as acções a serem tomadas.

2.3.3.2 Fundações de Arquitectura

O raciocínio envolve dois importantes processos: decidir qual o objectivo a atingir e como atingi-lo. O primeiro processo é deliberativo enquanto o segundo é de execução. De forma genérica, um agente num determinado momento no espaço e no tempo deve tomar uma decisão quanto à acção que vai executar. Esta decisão começa tipicamente na análise das opções disponíveis. Depois de serem escolhidas e agrupadas num conjunto de alternativas, o agente deverá dedicar-se a algumas. Estas opções escolhidas transformam-se em intenções, que determinam as acções do agente. Quando uma intenção é adoptada todo o processo futuro de raciocínio do agente ficará influenciado por esta decisão. Por exemplo, não irão ser seguidos caminhos inconsistentes com esta decisão, pois esta persiste até estar concluída ou haver bases que a impossibilitem. As intenções estão ainda intimamente relacionadas com as crenças, pois se o agente apresenta uma intenção então deve acreditar na sua possível concretização.

Podem destacar-se as seguintes características relacionadas com as intenções:

1. Intenções conduzem a planos de execução: para toda intenção existe um plano de concretização. Se este se tornar inviável outros deverão ser escolhidos;

2. Intenções influenciam deliberações futuras: os caminhos futuros a seguir não irão contra as bases desta escolha, da mesma forma que para se seguir uma intenção é necessário haver crenças que permitam esta hipótese e não a contrariem. Estas crenças assumidas vão de igual forma influenciar todo o processo futuro de raciocínio do agente;

3. Intenções persistem: uma intenção persiste até estar concluída, até haver bases que impossibilitem a sua concretização, ou até as condições iniciais para a escolha da intenção já não se verificarem.

Weiss [Wei99, p.56] refere ser difícil atingir um bom balanço entre estas propriedades. Pelos motivos previamente citados um agente deve desistir de algumas intenções e de tempo em tempo reconsiderá-las. No entanto esta reconsideração tem um custo tanto a nível de recursos computacionais como temporais. A natureza deste compromisso, entre grau de dedicação e reconsideração foi estudada por David Kinny e Michael Georgeff [KG91] num número de experiências com a *framework distributed MultiAgent Reasoning System* (dMARS). Em suma, diferentes tipos de ambientes requerem diferentes tipos de decisões estratégicas. Em ambientes mais estáticos um comportamento puramente

orientado a objectivos é adequado, enquanto em ambientes mais dinâmicos a habilidade para reagir a alterações através da alteração de intenções é mais importante.

Weiss [Wei99, p.57] descreve ainda sete componentes integrantes numa arquitectura BDI (ver figura 2.1):

- Um conjunto de crenças, representando a informação que o agente tem acerca do meio ambiente;
- Uma operação para revisão de crenças (*belief review function - brf*), que recebe o *input* perceptual e as crenças do agente e determina o novo conjunto de crenças;
- Uma operação para geração de opções (*options*), que determina as opções disponíveis para o agente (os seus desejos) com base nas suas crenças e intenções;
- Um conjunto de opções, representando as acções disponíveis para o agente;
- Uma operação de filtro (*filter*), que representa o processo deliberativo do agente, e determina as suas intenções com base nas crenças, desejos e intenções;
- Um conjunto de intenções, representando o foco corrente do agente;
- Uma operação para selecção de operações (*execute*), que determina a acção a realizar com base nas intenções.

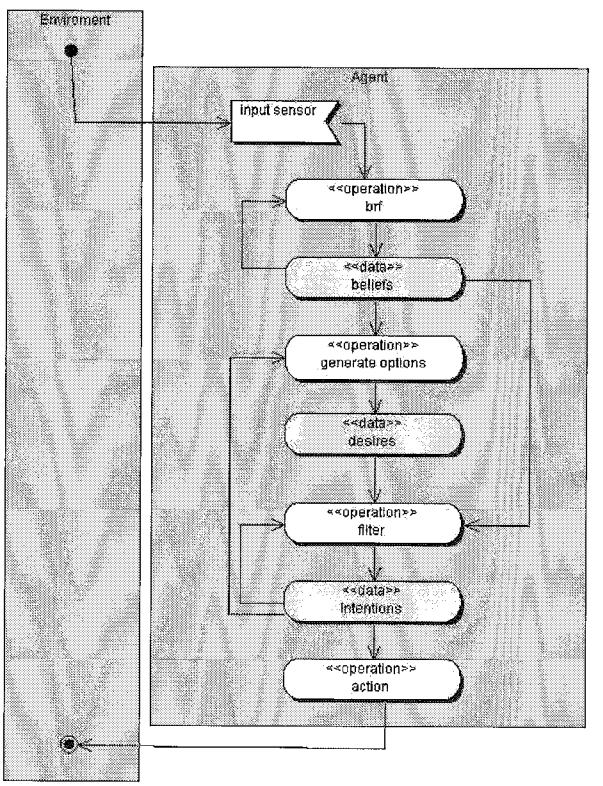

Figura 2.1: Diagrama de Actividades de uma Arquitectura BDI Genérica [Wei99, p.58].

Em forma de pseudo-código, estes componentes podem apresentar as seguintes dependências e fluxo:

```
public Action action(Input input){
  Beliefs beliefs = brf(beliefs, input);
```

```
  Desires desires = options(desires, intentions);
  Intentions newIntentions = filter(beliefs, desires, intentions);
  return execute(newIntentions);
}
```

Rao e Georgeff [RG95] introduzem também um modelo semelhante ao anterior, respeitando os conceitos teóricos introduzidos por Bratman[Bra87]. A arquitectura é apresentada numa perspectiva bastante prática, sendo uma versão simples do *Procedural Reasoning System* (PRS) [GL87], uma das primeiras soluções orientada a agentes baseada numa arquitecura BDI. O nível de abstracção usado é bastante inferior a muitas outras investigações, de forma a ser viável como sistema.

2.3.3.3 Conclusão

O modelo BDI é atractivo por diversas razões. É um modelo largamente estudado há vários anos, e apresenta um nível elevado de intuição pois o processo de decidir o que fazer e posteriormente como o fazer é comum ao quotidiano do ser humano, assim como as noções de crenças, desejos e intenções. A decomposição descrita acima e apresentada por Weiss torna ainda evidente quais os subsistemas em causa para construir um agente com modelação BDI.

Embora os conceitos introduzidos sejam de fácil acessibilidade e compreensão, a construção de um modelo deste tipo não é trivial. Existem várias arquitecturas BDI descritas por vários autores ([Wei99], [RG91], [RG95], [KGR96], [BRHL99], [GPP+99]), algumas outras implementadas e disponíveis (Jadex[13], PRS, o seu sucessor dMARS, *The Java Agent Kernel* (JACK), *The Java Agent Model* (JAM), entre outros), no entanto nenhuma vingou como *standard* a ser seguido. O estudo destas arquitecturas encontra-se fora de escopo desta dissertação e mais detalhes poderão ser encontrados na referências bibliográficas usadas.

Note-se ainda que a maioria destas plataformas disponíveis são apresentadas como parte integrante de uma *framework* de Agentes, situação que reduz drasticamente a seu nível de usabilidade e interoperabilidade. Seria interessante poder-se usar um componente mais genérico que implementasse um modelo BDI reutilizável por entre várias arquitecturas de agentes. Percebe-se a dificuldade em separar estes dois conceitos, tanto a nível técnico como a nível teórico, no entanto esta possibilidade existe e uma implementação poderia passar pela utilização de adaptadores ou *handlers* específicos para cada plataforma, prevalecendo todo o núcleo do componente. Um componente deste tipo traria um conjunto de benefícios imediatos não só para o modelo teórico mas também para a demonstração prática do problema em larga escala.

2.4 Reflexão

A tecnologia de agentes tem sido um tema muito discutido ao longo de muitos anos. A palavra "agente" foi e ainda é, alvo de muito uso e abuso. Devido a estes abusos, o conceito banalizou-se e deturpou-se. Ainda assim é um conceito importante e tecnologicamente promissor.

Provavelmente o agente mais conhecido dos últimos tempos seja o agente Smith, da trilogia Matrix. A sua popularidade e protagonismo cresceu com a evolução da personagem, um agente. Os agentes descritos na saga referida são representações visuais de programas que ajudam a manter a "matriz" num estado consistente. Não a controlam, apenas actuam como agentes de limpeza residindo nela permanentemente. No entanto

[13] Ver http://vsis-www.informatik.uni-hamburg.de/projects/jadex/

os agentes são "inteligentes", como se pode analisar a partir da expressão do Agente Smith quando refere a Morpheus que odeia aquele lugar. Apresentam um estado de consciência, raciocinam, e usam o poder de percepção para concretizarem a sua tarefas de forma autónoma, colaborando com outros quando necessário. Apesar de ser uma ideia futurista para utilização de agentes, os conceito aplicados são exactamente os mesmos referidos ao longo deste capítulo.

Actualmente foram usados agentes inteligentes no projecto eSTAR, onde os astrónomos usam agentes para controlar o telescópio de infravermelhos do Reino Unido, na observação da anã branca Nova. Um dos astrónomos refere *"The Agents can detect and respond to the rapidly changing universe faster than any human... [they] can be used to assist human observers, instead of replacing them entirely augmenting their abilities to do science quicker, faster, and more reliably"*. Estes dados são posteriormente enviados para telemóveis de telefones móveis de geração 2.5G e 3G, em formato de imagem, e distribuída informação para outros centros. É espectável que o uso desta tecnologia cresça num futuro próximo.

Dependendo da forma de utilização e dos estudos elaborados no futuro, os agentes poderão tornar-se parte integrante do nosso dia-a-dia. Com a definição de *standards* de comunicação, estes poderão assistir o ser humano em variadas tarefas, desde escalonamento de encontros, gestão de finanças pessoais, compras, e até entretenimento ou prazer. Esta segunda geração de agentes caracterizada pela capacidade comunicativa e cooperativa, provavelmente irá ser parte integrante do nosso futuro.

As futuras gerações serão ainda mais complexas. Talvez até ao ponto de se replicarem e criarem sub-agentes que comandarão. Provavelmente, e como dita a história do Homem, serão cada vez mais parecidos connosco e cada vez mais eficientes, e inteligentes. Estas habilidades certamente aproximarão, como nunca, os agentes da verdadeira inteligência artificial e também vida artificial.

2.5 Resumo

Ao longo deste capítulo é possível notar que não existe um consenso na definição de agente. Cada investigador define de acordo com a sua visão ou compreensão, resultando num conceito algo subjectivo. No entanto é interessante notar que existem várias semelhanças por entre as diversas descrições. Alguns conceitos são usados da mesma forma com o objectivo de descrever um agente. De entre estes conceitos destacam-se:

1. Autonomia: Capacidade de realizar acções de forma independente, sem necessidade de acompanhamento do utilizador. O agente deve por si só determinar a melhor solução e agir por sua iniciativa;

2. Adaptabilidade/Aprendizagem: Um agente adapta-se através da aprendizagem, melhorando o seu desempenho com base em experiências passadas;

3. Comunicação: Um agente deve ser capaz de comunicar com outros agentes ou humanos. Esta comunicação deve ser feita através de uma linguagem de comunicação expressiva, idealmente assemelhando-se a um discurso humano;

4. Cooperação/Social: Os agentes devem ser capazes de comunicar uns com outros, ou com humanos, de forma a cooperarem e realizarem tarefas complexas;

5. Responsivo/Reactivo: Um agente deve conseguir compreender o seu meio ambiente e ser capaz de realizar acções baseadas nestas percepções;

6. Mobilidade: A capacidade de um agente se mover de um sistema para outro de forma a aceder a recursos remotos, ou a outros agentes;

7. Personalidade/*Lifelike*: O objectivo de inserir personalidade num agente é criar a ilusão de sentimentos, emoções e interacção social;

8. Proactividade: Os agentes não se devem restringir somente a respostas, mas deve também apresentar comportamento oportunista e orientado a objectivos, com iniciativa quando apropriado.

Ao longo do capítulo foram também descritos alguns tipo de agentes, nomeadamente:

1. Agentes Colaboradores: Autonomia e cooperação são características que predominam neste tipo de agente. Tipicamente dispõem de capacidade social, poder de resposta e pro-actividade, sendo capazes de agir com racionalidade em ambientes multi-agentes e com constrangimentos de temporais;

2. Agentes Pessoais ou Interfaces: Realizam tarefas para o seu proprietário ou utilizador. Para tal, apresentam como características principais autonomia e aprendizagem. Este tipo de agente pode ser visto como um assistente pessoal que colabora com o utilizador no mesmo ambiente de trabalho;

3. Agentes Móveis: O agentes móveis constituem uma ferramenta de grande interesse para a maioria de aplicações que exigem um elevado grau de automatização. São processos computacionais capazes de percorrer vastas áreas. Interagem com hospedeiros remotos, armazenado informação para o utilizador, e retornando ao local da partida aquando a satisfação dos seus objectivos;

4. Agentes da *Internet* ou Informativos: Foram criados para lidarem com tarefas hoje impossíveis para humanos, o tratamento e gestão de grandes quantidades de informação. Diferenciam-se dos agentes colaboradores por serem definidos pelo que fazem, e não pelo que são. Existem ainda duas razões principais para se desenvolverem agentes informativos. A primeira e já referida necessidade de lidar com grandes quantidades de informação, e a segunda motivos económicos;

5. Agentes Reactivos: Também conhecidos por agentes reflexivos, reagem com regras estímulo-resposta ao estado do meio ambiente. Representam uma categoria especial de agentes que não possuí modelos simbólicos internos do meio em que se encontram, apresentando total incapacidade de previsão de acontecimentos. Como vantagem principal apresentam robustez e rapidez de resposta;

6. Agentes Híbridos: De forma a diminuir as deficiências e maximizar as qualidades, foram criados os agentes híbridos. Este agentes são uma combinação de dois ou mais tipo de agentes. A objectivo é conseguirem atingir um nível de eficiência superior à conseguida por um tipo de agente simples;

7. Sistemas de Agentes Heterogéneos: Um sistema heterogéneo de agentes, ao contrário dum híbrido refere-se a um sistema constituído com pelo menos dois tipos diferentes de agentes. Pode também conter agentes híbridos. A ideia principal que está por detrás deste tipo de sistema, é a interacção entre diferentes agentes, onde cada um está especializado em determinadas tarefas;

8. *Smart Agents*: Os agentes perfeitos de Nwana (ver 2.2.2.1).

Capítulo 3

Dynamic Logic Programming

Este capítulo apresenta o tema *Dynamic Logic Programming* (DLP), estudando a representação dinâmica de conhecimento e uma linguagem que a permita por em prática. Toda teoria apresentada neste capítulo encontra-se descrita em [ALP+98], [APPP99], [AP02] e [APP+99a], devendo ser seguidas para mais detalhes.

3.1 Introdução

A representação de conhecimento é foco de investigação na área de inteligência artificial desde os seus primórdios (na época de 50). Uma das aproximações é caracterizada pelo "baixo nível" nas estruturas de dados e respectiva manipulação. Para o mesmo efeito são usadas construções posteriores, e as teorias correspondentes são por vezes um pouco de filosofia simples. As aproximações mais orientadas ao nível teorico são baseadas na lógica ou teoria de decisão. A representação de conhecimento formaliza e organiza o conhecimento.

Um tipo de representação largamente usado são as regras. Uma regra é composta por duas partes, uma IF e outra THEN (também chamada de condição ou acção). A parte IF lista um conjunto de condições numa combinação lógica. A peça de conhecimento representada por uma regra é relevante para o raciocínio a ser desenvolvido se a parte IF de uma regra poder ser satisfeita. Consequentemente a parte THEN pode ser concluída, ou a acção desencadeada. Os sistemas[1] cujo conhecimento é representado na forma de regras são chamados de *rule-based systems.*

A representação de conhecimento e o "raciocínio" lidam com os aspectos formais da própria representação e modelação de problemas, assim como trabalham estes modelos posteriormente. O ponto de equilíbrio é o balanço entre a expressividade da representação, e a complexidade associada aos algoritmos ditos inteligentes.

3.2 Representação, Regras e DLP

Em DLP, regras novas ou mais específicas podem ser adicionadas no final de uma sequência, não interferindo com o conhecimento anterior ou mais específico. Uma sequência pode ser vista como um resultado de um programa P_1, actualizado por um programa P_2, até um programa P_n $(P_1 \oplus \ldots \oplus P_n)$. O papel da programação dinâmica é certificar-se que estas novas regras adicionadas estão "*in force*", e que regras anteriores ou mais específicas estão ainda válidas (por inércia), i.e. mantidas até entrarem em conflito com as novas adicionadas [AP02, p.11].

[1] *Expert systems.*

A tecnologia DLP pode ser interpretada como uma *framework* usada para modelar a evolução dum programa lógico através de uma sequência de actualizações [AP02, p.11].

Para representar informação negativa em programa lógicos e nas suas actualizações, a DLP permite a presença da negação por defeito nas cabeças das regras. É importante perceber-se porquê, nas configurações das actualizações, generalizando a linguagem para permitir a negação por defeito na cabeça das regras é mais adequado que introduzir a negação explícita em programas (tanto na cabeça, como no corpo de regras) [AP02, pp.11-12].

A diferença entre a negação explícita ($\neg$, ver regra 3.1) e a negação por defeito (*not*, ver regra 3.2) é fundamental quando a informação sobre um átomo A não pode ser assumida como completa. Com estas circunstâncias, a primeira significa que há evidências de A ser falso, enquanto a segunda significa que não há evidência de A ser verdadeiro. Para o caso da remoção de regras, é desejável o último caso. Por outras palavras, *not* A significa que A é removido se o corpo da regra for provado. Apagar A significa que não é verdadeiro, não necessariamente falso [AP02, p.12].

$$\neg A \leftarrow Cond' \tag{3.1}$$

$$not\ A \leftarrow Cond' \tag{3.2}$$

A DLP não fornece por si só uma linguagem para especificar (ou programar) alterações a programas lógicos. Se o conhecimento já está representado em programas lógicos, os programas dinâmicos simplesmente representam a evolução do conhecimento. Visto que os programas lógicos descrevem estados do conhecimento, podem igualmente representar a transição de estados do conhecimento. É natural associar a cada estado um conjunto de regras de transição para o próximo estado. Como resultado irá ser construído um conjunto com sequências de transições de estados e regras[2] .

3.3 Dynamic Knowledge Representation

Um dos requisitos principais do formalismo usado para representar o conhecimento é a habilidade para lidar com a evolução do conhecimento.

A DLP foi proposta por [ALP+98] como uma possível solução para este requisito evolucional. Define como uma base de conhecimento pode ser actualizada por outra, resultando numa nova base de conhecimento.

Especificamente, dada uma base de conhecimento KB, e uma outra para a actualizar KB', é possível obter uma nova base de conhecimento actualizada $KB^* = KB \oplus KB'$ que constitui a actualização de KB por KB'. De forma a traduzir um significado declarativamente claro e facilmente verificável para uma base de conhecimento actualizada, em [ALP+98] é realizada uma caracterização semântica da base de conhecimento actualizada $KB \oplus KB'$.

3.3.1 LUPS - Language of Updates

Em DLP uma base de conhecimento evolui de estado em estado como resultado de actualizações ao conhecimento. Dado o estado corrente KS, o seu sucessor é $KS' = KS \oplus$

[2]Na secção 3.3.1 é apresentada uma linguagem para especificar actualizações a programa lógicos: *Language of dymamic UPdateS* (LUPS)[APPP99]. A linguagem LUPS modela os estados e as suas transições de forma declarativa.

KB é obtido como resultado da ocorrência de um conjunto não vazio de actualizações simultâneas, representado pela base de conhecimento KB.

No entanto, as actualizações dinâmicas de conhecimento não fornecem qualquer linguagem que permita a especificação de alterações aos estados de conhecimento. Em [APP+99a] foi descrita uma linguagem totalmente declarativa para actualizações de conhecimento chamada *Language of dymamic UPdateS* (LUPS), que descreve as transições entre estados de conhecimento consecutivos KS_n. Consiste em comandos de actualizações que especificam quais as actualizações que devem ser aplicadas a um determinado estado de conhecimento KS_n com o objectivo de obter os consequente estado KS_{n+1}. Desta forma, os comandos de actualização permitem determinar implicitamente a base de conhecimento actualizada KS_{n+1}. A linguagem LUPS pode ser vista como uma linguagem para representação dinâmica de conhecimento.

O comando de actualização mais simples consiste na adição de uma regra ao estado corrente e apresenta a forma: $assert\ (L \leftarrow L_1, \ldots, L_k)$

De forma geral, a adição de uma regra a um estado pode depender da validação de algumas de pré-condições nesse mesmo estado. Para o permitir, o comando *assert* em LUPS apresenta uma forma mais geral:

$$assert\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{3.3}$$

O significado do comando assert é que se as pré-condições $L_{k+1}, \ldots, L_m$ forem verdadeiras no presente estado, então a regra $L \leftarrow L_1, \ldots, L_k$ deve permanecer verdadeira no estado sucessor. Normalmente as regras adicionadas são "inérciais". Por outras palavras, permanecem "*in force*" a partir deste estado, por inércia, até removidas ou destituídas por outras actualizações.

No entanto em alguns casos a persistência de regras por inércia não deve ser assumida, ou tomada como opção. Note-se o exemplo de uma acção *request exit*. Provavelmente é um evento que ocorre uma só vez e não deve persistir por inércia depois do estado sucessor. Como tal, o comando assert permite o uso da keyword *event*, indicando que a *regra* adicionada é *não inércial*.

$$assert\ event\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{3.4}$$

Os comandos de actualizações (em vez das regras que introduzem) podem ser eventos não persistentes ou podem permanecer até posteriormente cancelados. Tais comandos de actualizações persistentes (chamados *leis de actualização*) são descritos da seguinte forma:

$$always\ [event]\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{3.5}$$

Note-se que a *keyword event* é opcional e define se o comando deve persistir ou ser de um estado apenas.

Para cancelar comandos persistentes, usa-se:

$$cancel\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{3.6}$$

Para se apagarem regras existe um comando de *retraction*, que sujeito a determinadas pré-condições $L_{k+1}, \ldots, L_m$, a regra $L \leftarrow L_1, \ldots, L_k$ pode ser removida:

$$retract\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{3.7}$$

A diferença principal entre o comando *cancel* e *retract* é que o primeiro é usado para cancelar um comando persistente, ou uma lei *always*, enquando o segundo é usado para

cancelar um *assert*.

Podem ainda ser feitas *queries* a qualquer estado da base de conhecimento KS_q em que o resultado é verdadeiro se a conjunção de todos os literais for válida para o estado KS_q:

$$\textbf{holds } B_1, \ldots, B_k, not\, C_1, \ldots, not\, C_m \textbf{ at state } q? \tag{3.8}$$

Nesta equação o q é um identificador do estado do conhecimento e pode ser definido por um inteiro onde o número zero representa o estado inicial. O *not* significa a negação por defeito e o predicado *holds* verifica se os literais pertencem ao modelo da base de conhecimento KS_q.

A linguagem LUPS apresenta uma "semântica procedimental e declarativa"[3] [ALP+98], para que as actualizações não tenham somente um significado declarativo mas que também possam ser implementadas. A semântica procedimental da linguagem LUPS é obtida através da sua tradução para um programa lógico escrito numa meta-linguagem [4].

3.4 Resumo

A programação em lógica, foi considerada nos anos 90 como uma poderosa *framework* orientada para a investigação teórica da representação de conhecimento. Fornece ainda uma base conceptual clara e uma semântica bem definida. No entanto, o problema de alterar e integrar informação com representação de conhecimento não atraiu muita atenção. A introdução da DLP refrescou estes conceitos e alargou a dimensão desta área tecnológica.

Em DLP, regras novas ou mais específicas podem ser adicionadas no final de uma sequência, não interferindo com o conhecimento anterior ou mais específico. Uma sequência pode ser vista como um resultado de um programa P_1, actualizado por um programa P_2, até um programa P_n $(P_1 \oplus \ldots \oplus P_n)$. O papel da programação dinâmica é certificar-se que estas novas regras adicionadas estão consistentes, e que regras anteriores ou mais específicas estão ainda válidas - por inércia.

Em [APP+99a] foi descrita uma linguagem totalmente declarativa para actualizações de conhecimento, chamada LUPS (*Language of UPdateS*), que descreve as transições entre estados de conhecimento consecutivos KS_n. Consiste em comandos de actualizações que especificam quais as actualizações que devem ser aplicadas a um determinado estado de conhecimento KS_n, com o objectivo de obter os consequente estado KS_{n+1}. Desta forma, os comandos de actualização permitem determinar implicitamente a base de conhecimento actualizada KS_{n+1}. A linguagem LUPS pode ser vista como uma linguagem para representação dinâmica de conhecimento.

[3] *Declarative and Procedural Semantics.*

[4] A tradução do LUPS para XSB-Prolog está disponível em http://centria.di.fct.unl.pt/~jja/updates/lups.p.

Capítulo 4

Arquitecturas Orientadas a Serviços

O curto ciclo de vida dos produtos, e a elevada competição existente ao nível comercial, faz com que as empresas na área das tecnologias de informação se foquem em criar ambientes flexíveis; assegurar uma posição estratégica para maior inovação; responder de forma rápida a alterações através da redução do *time to market* de novos serviços; e por fim diminuir os custos associados a estes processos. Esta flexibilidade pode ser concretizada através das arquitecturas orientadas a serviços, um paradigma de desenho baseado numa colecção de serviços *loosely coupled* e reutilizáveis. O presente capítulo descreve de forma sucinta este estilo arquitectural, assim como a sua envolvente tecnológica.

4.1 Introdução

Uma arquitectura orientada a serviços, ou *Service Oriented Architecture* (SOA), consiste num estilo arquitectural que permite definir arquitecturas de integração orientadas ao conceito de serviço, promovendo a utilização de mecanismos de integração baseados em *standards*, de forma a possibilitar a integração de processos de negócio. Um serviço é portanto uma implementação de uma ou mais funções de negócio bem definidas, que podem ser consumidas/invocadas por clientes (aplicações com características diversas).

Uma arquitectura com este estilo descreve um tipo de aplicação distribuída usado há vários anos. Um exemplo deste estilo é a tecnologia *Common Object Request Broker Architecture* (CORBA), que está em uso desde a década de oitenta. A tecnologia *Distributed Component Object Model* (DCOM), da Microsoft é outro exemplo de uma tecnologia que fornece uma *Interface Definition Language* (IDL) e mecanismos de distribuição *tightly coupled* que permite invocações remotas ou locais. No entanto, em contraste com a tecnologia CORBA e DCOM, uma SOA moderna baseada em *web services* constituí um modelo *loosely coupled*, fornecendo a independência de plataformas, linguagem, transporte e formato de mensagem.

Uma SOA permite portanto atingir um nível de flexibilização não possível anteriormente, uma vez que:

- Os serviços são componentes de *software* com interfaces bem definidas e independentes da implementação. Um aspecto importante desta arquitectura é a separação clara da interface da sua implementação. E os serviços são invocados por clientes que não estão interessados com a forma como os serviços se executam em resposta ao seu pedido;
- Os serviços são *self-contained* (realizam tarefas concretas) e são desacoplados;

- Os serviços podem ser descobertos dinamicamente;
- Podem ser criados serviços compostos pela orquestração ou composição de outros serviços.

Uma SOA usa o paradigma *find-bind-execute*, como apresentado na figura 4.1. Neste paradigma os fornecedores dos serviços procedem ao registo público do serviço. Este registo é usado pelos clientes para pesquisar serviços com determinados critérios. Se o registo tiver tal serviço, ele disponibiliza o contrato do serviço ao cliente com o endereço do *endpoint* para tal serviço.

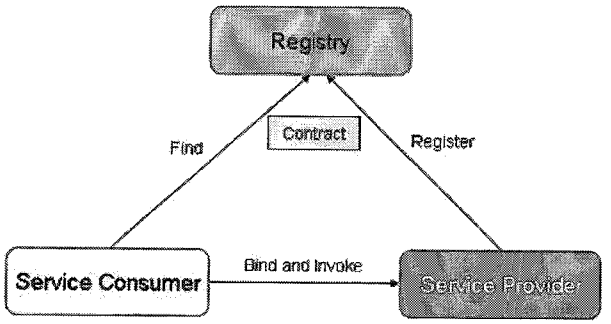

Figura 4.1: Paradigma *Find, Bind and Execute.*

Os serviços são o elemento base de uma aplicação SOA. Enquanto qualquer funcionalidade poder ser convertida num serviço, o desafio passa por definir a interface do serviço que se encontre no nível certo de abstracção. Porém, uma SOA é mais do que um conjunto de serviços, é também composta por um conjunto de princípios. A interoperabilidade consiste no mais importante destes princípios, sendo nuclear a este estilo arquitectural e atingindo-se pela utilização de *standards*. Dos standards relacionados com *web services* destacam-se a invocação remota através de *Simple Object Access Protocol* (SOAP)[1], a descrição das interfaces de serviços utilizando *Web Services Description Language* (WSDL)[2] e a sua possível pesquisa e recuperação via *Universal Description, Discovery and Integration* (UDDI)[3]. No entanto existem abordagens alternativas que não introduzem *standards* novos e reutilizam os já existentes, como a *Representational State Transfer* (REST) [Fie00].

Uma arquitectura deste tipo apresenta-se como solução ideal para integrar clientes em ambiente heterogéneos, disponibilizando um ponto de acesso comum para todos. Esta solução pode ainda ser integrada num bus de serviços - *Enterprise Service Bus* (ESB) - podendo fornecer e registar os serviços desenvolvidos, assim como usar outros mais antigos de forma isolada ou coreografada.

4.2 Serviços

4.2.1 Características

Um serviço é um componente de *software* com um significado funcional importante, que normalmente encapsula um conceito de negócio. Apresenta as seguintes características:

[1]Consultar http://www.w3.org/TR/soap/.
[2]Consultar http://www.w3.org/TR/wsdl.
[3]Consultar http://www.uddi.org.

Contracto O contracto de um serviço fornece uma especificação informal do objectivo do serviço, da sua funcionalidade, constrangimentos e utilização. Este tipo de especificação pode variar, dependendo do tipo de serviço em causa. Um elemento importante do contracto é a definição formal da interface, tal como *Interface Definition Language* ou *Web Services Description Language*. Esta especificação embora não obrigatória, adiciona um importante benefício ao nível da abstracção, independência tecnológica, protocolos, entre outros. No entanto um contracto é algo mais que uma especificação formal, pois pode impor um detalhe semântico na funcionalidade e parâmetros não especificada na *Interface Definition Language* ou *Web Services Description Language*.

Interface A funcionalidade do serviço é exposta através da interface do serviço aos seus clientes;

Implementação A implementação do serviço fornece a lógica de negócio e acesso aos dados apropriados, sendo a realização técnica do contrato do serviço;

4.2.2 Identificação e Tipos de Serviços

Como a peça chave deste estilo arquitectural são os serviços, a sua identificação é a primeira tarefa na modelação de uma solução orientada a serviços.

4.2.2.1 Metodologias

O primeiro passo a ser tomado passa por decidir se o processo de identificação é puramente baseado na compreensão das operações, posteriormente agregadas em serviços, ou se os serviços já estão identificados e as operações lhes são adicionadas quando identificadas. Existem portanto as seguintes técnicas:

Service-First Esta técnica é comum ao desenvolvimento *object-oriented* e *component-oriented*, onde classes de negócio são identificadas primeiro. Posteriormente são analisadas as colaborações entre objectos, e as responsabilidades de um objecto (operações) são identificadas e adicionadas às classes. Da mesma forma, os serviços podem ser identificados através do mesmo processo;

Operation-First Esta técnica pode ser considerada como mais intuitiva em relação à anterior. Um serviço é considerado uma entidade diferente de uma classe, objecto ou componente. Podem gerir um conjunto de recursos, no entanto a relação serviço/recurso é substancialmente diferente da relação classe/objecto. Como tal, esta técnica promove uma "identificação tardia" de serviços através da agregação de um conjunto de operações em grupos lógicos.

Qualquer destas abordagens é independente das metodologias abaixo descritas.

Top-Down, Orientado a Processos de Negócio/Casos de Uso

Um processo de negócio está focado nas tarefas realizadas por determinados *roles*, ou recursos numa organização, para concretizar essa mesma tarefa. Pode ser visto como um conjunto de tarefas ordenadas, possivelmente decompostas em sub-processos. Geralmente o objectivo é expor ou fornecer uma nova funcionalidade ao exterior.

Numa SOA, um serviço é identificado com um nível de granularidade similar. É assumido que as operações numa especificação de serviço correspondem a uma transacção atómica num processo de negócio - *Top-Down, Business Process Driven.* É portanto

normalmente assumido que uma vez identificados os serviços no processo de negócio, estes possam ser directamente implementados tal como descritos no processo ou fluxo. No entanto é importante ter em conta os requisitos não funcionais (segurança, qualidade de serviço, entre outros), que podem afectar o tipo de serviço a ser disponibilizado. A transformação de um processo num conjunto de especificações iniciais de serviços (tarefas mapeadas para operações) é considerada uma boa prática inicial, no entanto requer análise posterior antes do desenho que detalha a implementação.

Como alternativa à orientação a processos de negócio, pode-se usar o modelo de casos de uso para auxiliar a identificação de serviços - *Top-down, Use case Driven.* Este modelo mantém um conjunto de objectivos de negócio, que facilmente auxiliam a identificação de serviços. É igualmente fácil enumerar para um serviço quais os objectivos de negócio com que ele contribui. De forma prática, um caso de uso pode ser mapeado para uma operação. Para poder agregar estas operações em serviços, é importante refinar o modelo de casos de uso para um modelo de análise de negócio, onde é possível perceber a colaboração entre entidades.

Bottom-up, Exposição de Recursos

É importante ter em conta que poucas são as soluções orientadas a serviços que não consideram aplicações já existentes, assim como as suas funcionalidades. Muitas das vezes é necessário interagir com estas funcionalidades, e tão importante como reutilizá-las, é identificá-las e catalogá-las. Numa SOA, existe um conjunto de caminhos para integrar funcionalidades existentes com novos serviços:

1. *Wraping* da funcionalidade como um serviço: Neste caso a função inicial é deixada intacta, usando-se ferramentas ou middleware para as expor como serviços;

2. *Wraping* e substituição da funcionalidade por um serviço: Neste caso são realizados os mesmos passos descritos no ponto anterior, no entanto usa-se a especificação do serviço resultante para voltar a desenvolver o serviço inicial numa data futura, substituindo a implementação original por uma mais actual;

3. Uso de Adaptadores: Por vezes pode ser complicado realizar o *wrap* de uma funcionalidade e expô-la como um serviço, no entanto pode ser possível expor a funcionalidade de uma forma harmoniosa através de um adaptador (p.e. através da tecnologia *J2EE Connector Architecture* (JCA)). Desta forma os novos serviços podem interagir com a referida funcionalidade através desta camada adaptadora;

4. Integrar a funcionalidade num serviço: Este caso indica que por vezes pode ser viável um serviço (novo ou não) aceder directamente à funcionalidade em causa, usando a função com um componente lógico da sua implementação.

As duas primeiras soluções são consideradas mais limitativas se não se tiver em conta uma actualização da interface, para bem da interoperabilidade entre serviços. Uma nova interface poderá requerer um novo processo de análise possivelmente culminando em novos campos de *input/output* - ou tipos diferentes. Poderá ainda exigir algum código de transformação de tipo de dados. Por outro lado, a última opção poderá ser a mais flexível por não expor a funcionalidade directamente, sendo portanto uma solução muito semelhante à primeira e segunda com actualização da interface.

Rule Driven

Algumas soluções de software baseiam-se fortemente na utilização de regras de negócio para trabalhar e recuperar conteúdos, podendo estas regras evoluir para além da aplicação lógica principal. A partir do modelo de análise de negócio, incluindo entidades e regras, é possível definir serviços que encapsulem estas regras de negócio, tornando-as externas do resto da lógica da solução. Em vários casos, regras complexas são agregadas em conjuntos de regras, podendo estas estar mais equiparadas à granularidade de um serviço.

4.2.2.2 Heurística

A identificação de serviço é processo bastante complexo, e para uma correcta identificação é importante o leitor distinguir claramente entre um *proxy*, uma *façade*, e um processo de negócio, do ponto de vista granular e funcional. No final todos serão expostos como um serviço, no entanto o processo de criação é substancialmente diferente. Como abordado na secção anterior (ver metodologia *Bottom-up*, Exposição de Recursos), numa SOA ou numa arquitectura de integração normalmente existe a reutilização de muitas funcionalidades já disponíveis. Estas funcionalidades devem ser disponibilizadas no canal de informação por *wrappers*, ou *proxies* que expõem as funcionalidades dos serviços originais, delegando-lhes o trabalho. Podendo existir nestes *proxies* algum tratamento de dados de *input/output*, podem caracterizar-se como objectos simples e agnósticos do ponto de vista de negócio.

Do conjunto de funcionalidades existentes e reutilizáveis, é importante saber decidir quais destas devem ser promovidas a serviços. Num conjunto de novos requisitos funcionais, a solução pode passar por reutilizar vários destes serviços. Se do ponto de vista funcional, os serviços que integram uma nova funcionalidade não apresentam características de reutilização, por serem demasiado específicos ou por não conterem grande lógica de negócio, deve ser ponderada a criação de um serviço tipo *façade*, i.e. um serviço que usa outras funcionalidades disponibilizando-as numa nova cara, numa nova interface. Este tipo de opção deve ser analisada com cuidado, de forma a evitar alterações e remendos futuros. No entanto como características principais benéficas destaca-se o desempenho ao nível interno, pois evitará ser construído através de um processo de negócio com várias operações WSDL/SOAP. Este tipo de decisão passa muito pela existência de requisitos não funcionais.

Ao nível mais alto situam-se os processos de negócio. Numa SOA bem definida, a probabilidade de este tipo de "serviço" predominar é elevada, promovendo fortemente a reutilização de outros serviços. Qualquer que seja o fluxo identificado na fase de análise funcional, que tenha colaboradores bem identificados e orientados ao nível do negócio, este deve ser mapeado para um processo de negócio onde as operações atómicas serão elevadas a funcionalidades, por sua vez agregadas em *web services*. No final existirá uma coreografia entre serviços e operações gerida e monitorizada num *bus*.

4.3 *Enterprise Service Bus*

4.3.1 Definição

Várias tecnologias de integração - como SOAs, *Enterprise Application Integration* (EAI), *Business-to-Business* (B2B), MAS, e *web services* - tentam solucionar o desafio inerente à componente de integração, ao mesmo tempo melhorar os resultados técnicos e incrementar o valor de negócio associado. O *Enterprise Service Bus* (ESB) é a mais bem

sucedida destas iniciativas.

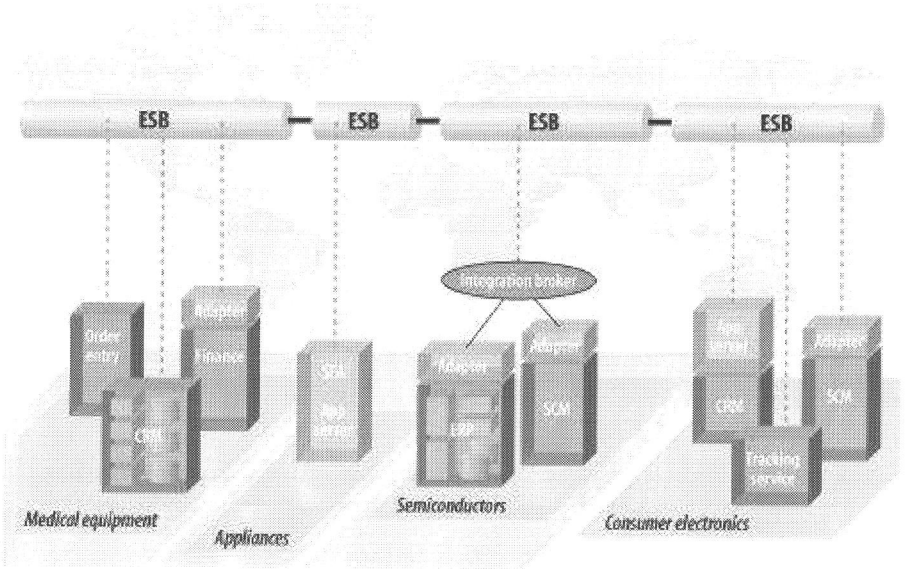

Figura 4.2: ESB, ambiente autónomo e federado [Cha04].

O conceito inerente ao ESB é uma aproximação à integração que fornece os alicerces de uma rede de integração *loosely coupled*, altamente distribuída e que escala muito além dos clássicos *brokers* EAI baseados em *hub-and-spoke*. Um ESB é uma plataforma de integração baseada em *standards*, que combina *messaging*, *web services*, transformação de dados, reencaminhamento inteligente e fiabilidade, de forma a coordenar toda uma interacção de diversas aplicações dentro ou fora dum ambiente empresarial, com transacionalidade, segurança e integridade.

4.3.2 Característica

Chappell [Cha04], identifica as seguintes características principais de um ESB:

Pervasiveness Um ESB pode ser adaptado para preencher as necessidades de um projecto de integração genérico;

Distribuição Elevada Componentes de integração *loosely coupled* podem ser instalados em vários pontos geográficos do *bus*, sendo que permanecem acessíveis como serviços para qualquer outro ponto do *bus*;

***Deployment* Selectivo** Adaptadores, Serviços e *routing* baseado em conteúdos podem ser instalados de forma selectiva quando sejam necessários;

Segurança e Confiança Todos os componentes que comunicam pelo *bus* podem usar os mecanismos por este fornecido, tal como garantia de entrega de mensagens , integridade transaccional, e comunicações seguras;

Orquestração e Processos de Negócio Um ESB permite fluxo de dados por várias aplicações e serviços que estejam ligados ao *bus*, seja a nível local ou remoto;

Ambiente Autónomo e Federado Um ESB apresenta autonomia local a um nível "departamental", no entanto é possível integrar-lo num ambiente de integração com maior dimensão. Por outras palavras, é possível existir um conjunto de ESBs

ligados remotamente, como uma federação de ESBs, sendo que a autonomia local de cada não é invadida;

Adopção Incremental Cada projecto individual pode ser construído de forma faseada de forma a criar uma rede de integração cada vez maior, que pode ser gerida remotamente e em qualquer lado do *bus*[4];

XML como Tipo Nativo Toda a representação de dados no *bus* é realizada através de *eXtensible Markup Language* (XML);

Monitorização e Configuração Remota É possível monitorizar qualquer elemento do *bus*, ou até o próprio *bus* através de *Business Activity Monitoring* (BAM);

Integração Baseada em *Standards* A conectividade é realizada através de componentes *Java Platform, Enterprise Edition* (Java EE), como *Java Message Service* (JMS) ao nível de mensagens, e JCA para conexão a adaptadores de aplicacionais (ver figura 4.3).

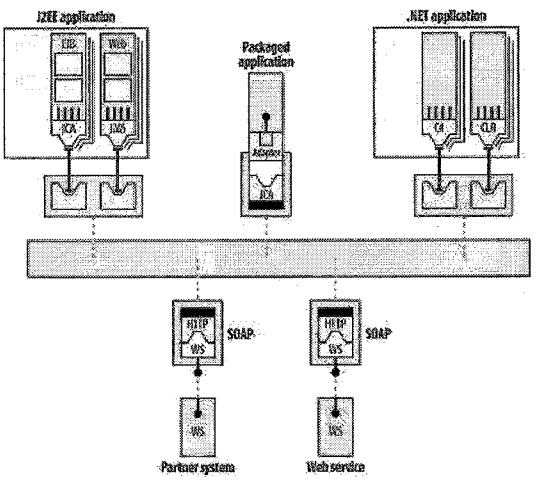

Figura 4.3: ESB integrando uma conjunto variado de tecnologias [Cha04].

4.4 Resumo

O presente capítulo apresentou de forma sucinta as SOAs e suas características principais. Foi possível perceber que os serviços, actualmente na forma de *web services*, juntamente com as suas funcionalidades de negócio formam o elemento base de uma SOA. Porém, uma SOA é mais do que um conjunto de serviços, é também composta por um conjunto de princípios. A interoperabilidade consiste no mais importante destes princípios, sendo nuclear a este estilo arquitectural e atingindo-se pela utilização de *standards*. Dos *standards* relacionados com *web services* destacam-se a invocação remota através de SOAP, a descrição das interfaces de serviços utilizando WSDL e a sua possível pesquisa e recuperação via UDDI.

Um aspecto igualmente importante dos *web services* e das SOAs é a sua ubiquidade suportada por um conjunto alargado de vendedores. Este facto provém das vantagens inerentes ao uso deste estilo arquitectural:

[4] A implementação de uma solução análoga à da figura 4.2 poderia passar por quatro fases, cada uma associada a um *bus* de informação que seria integrado com os restantes numa fase final.

- Flexibilidade no desenho de *software*;
- Reutilização de componentes de negócio;
- Interoperabilidade e integração;
- Facilidade de integração de novos processos de negócio.

Como o sucesso de uma SOA parte das suas fundações, foi focada a importância do processo de identificação de serviços, e descrita uma metodologia para os identificar e iniciar o seu desenho, tendo em conta serviços novos, reutilização de antigos, e coreografia de novos.

Uma arquitectura deste tipo apresenta-se como solução ideal para integrar clientes em ambiente heterogéneos, disponibilizando um ponto de acesso comum para todos. Esta solução pode ainda ser integrada num bus de serviços - ESB - podendo fornecer e registar os serviços desenvolvidos, assim como usar outros mais antigos de forma isolada ou coreografada.

Parte II

Solução

Capítulo 5

Análise Funcional

O presente capítulo documenta a fase de análise funcional e início do processo de desenvolvimento de *software*.

5.1 Introdução

O objectivo deste capítulo é fornecer uma visão e análise funcional do sistema, descrevendo o primeiro e segundo modelo do Processo Unificado [JBR99]: o modelo de casos de uso e o modelo de análise. O primeiro detalha os casos de uso e respectivas relações com os actores do sistema, enquanto o segundo refina o primeiro com maior detalhe, e descreve o comportamento do sistema com um conjunto de objectos.

A solução proposta irá apresentar as bases funcionais de uma arquitectura de agentes num ambiente *web*, sendo enquadradas num contexto de IR [BYRN99]. Os utilizadores desta plataforma serão clientes de um sistema com que interagem e trabalham a informação. Poderão interagir com o sistema de diversas formas, deste a autenticação, sumarização, pesquisas avançadas, até à administração do sistema lógico, podendo eventualmente influenciar os resultados.

O sistema desenvolvido foi nomeado "Susy". Doravante será usado este nome como referência à aplicação proposta e à plataforma em si.

5.2 Solução Proposta

Esta secção resume as principais funcionalidades do sistema, identificando os utilizadores que têm acesso a cada um delas. Pretende-se assim oferecer uma visão resumida do ponto de vista funcional, acompanhada duma especificação textual simples. O detalhe e realização destas funcionalidades será abordado em capítulos posteriores.

O modelo funcional da plataforma Susy está divido nos seguintes casos de uso, mapeados para dois tipos de actores (utilizador e administrador):

- **UC1.** Autenticação;
- **UC2.1** Pesquisa Simples;
- **UC2.2** Pesquisa Inteligente;
- **UC2.3** Pesquisa Interactiva;
- **UC3.** Sumarizar Documentos;
- **UC4.** Consultar Mensagens;

- **UC5.1** Gestão de Crenças;
- **UC5.2** Gestão das Base de Conhecimento.

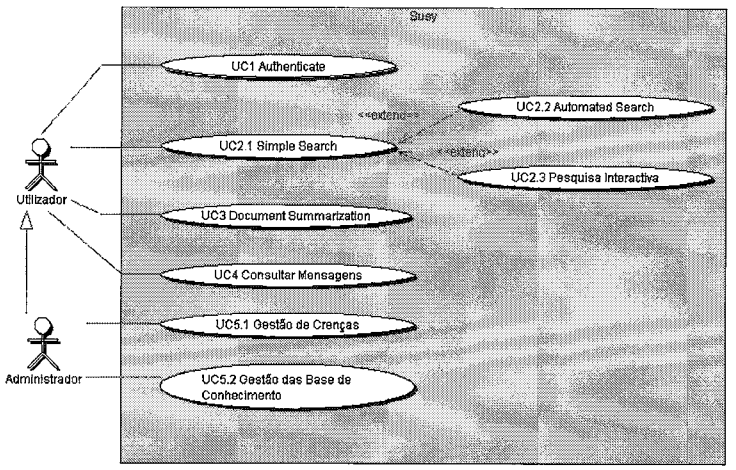

Figura 5.1: Casos de uso.

5.2.1 Autenticação

5.2.1.1 Descrição

Nome: UC1 Autenticação

Versão: 1.0

Autor: José Palmeiro

Última Actualização: 14/11/2005

Actores: Utilizador

Resumo: Processo de autenticação de um utilizador.

5.2.1.2 Fluxos

Inicialização: Este caso de uso é invocado sempre que um utilizador deseje entrar no sistema.

Autenticação de um Utilizador: Um utilizador para aceder ao sistema necessita de credenciais válidas. Para tal deve autenticar-se no sistema através da introdução do seu nome de utilizador e respectiva palavra chave. Se este tuplo for verdadeiro o utilizador será redireccionado para a página principal do sistema, onde possui um conjunto de opções disponíveis. Os diagramas de actividade descrito nas figuras 5.3 e 5.4 descrevem o presente fluxo.

Finalização: Este caso de uso é finalizado quando o utilizador for autenticado com sucesso ou com insucesso.

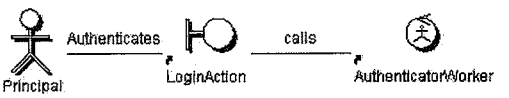

Figura 5.2: UC1 Autenticação - Diagrama de Robustez.

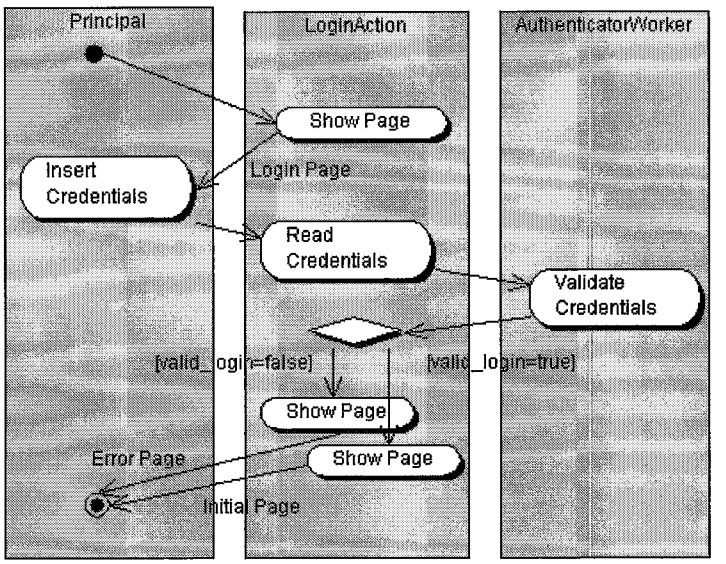

Figura 5.3: UC1 Autenticação - Diagrama de Actividade.

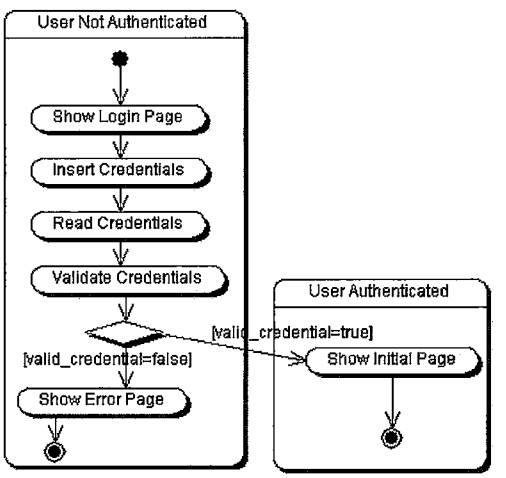

Figura 5.4: UC1 Autenticação - Diagrama de Actividade (2).

5.2.2 Pesquisa

5.2.2.1 Descrição

Nome: UC2 Pesquisa

Versão: 1.0

Autor: José Palmeiro

Última Actualização: 05/01/2006

Actores: Utilizador

Resumo: Esta funcionalidade permite que um utilizador autenticado realize pesquisas.

5.2.2.2 Fluxos

Inicialização: Este caso de uso é invocado sempre que um utilizador realize uma das pesquisa descritas nos restantes fluxos.

UC2.1 Pesquisa Simples: Uma pesquisa simples representa uma recuperação de informação baseada apenas nos dados introduzidos pelo utilizador. O diagrama de actividade descrito na figura 5.5 descreve o presente fluxo.

UC2.2 Pesquisa Automatizada: Uma pesquisa automatizada representa uma recuperação de informação baseada nos dados introduzidos, e no conhecimento previamente adquirido pelo sistema através da interacção com o utilizador. Com esta pesquisa o utilizador poderá beneficiar de novos resultados derivados de relações introduzidas e aprendidas pelo sistema, que poderão escalar novos conjuntos de resultados. Se o sistema não tiver dados disponíveis sobre os termos da pesquisa, irá automaticamente recuperar relações gramaticais (sinónimos, hiperónimos e hipónimos) que usará para pesquisa associadas. O diagrama de actividade descrito na figura 5.6 descreve o presente fluxo.

UC2.3 Pesquisa Interactiva: Uma pesquisa interactiva distingue-se da anterior pela interacção entre o utilizador e sistema, dividindo-se em dois passos principais. No primeiro o sistema é encarregue de fornecer ao utilizador relações gramaticais sobre os termos da pesquisa. O utilizador deve posteriormente seleccionar quais as relações interessantes e com base nestas o sistema realizará novas pesquisas. Todas as relações seleccionadas pelo utilizador serão armazenadas para posterior uso. O diagrama de actividade descrito na figura 5.7 descreve o presente fluxo.

Finalização: Este caso de uso é finalizado quando forem apresentados os resultados de uma pesquisa ou existam erros.

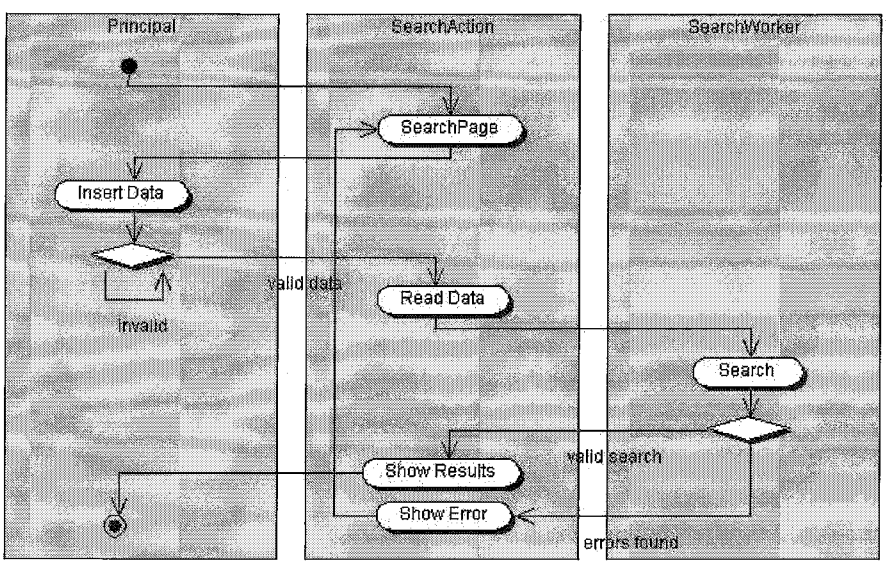

Figura 5.5: UC2.1 Pesquisa Simples - Diagrama de Actividade.

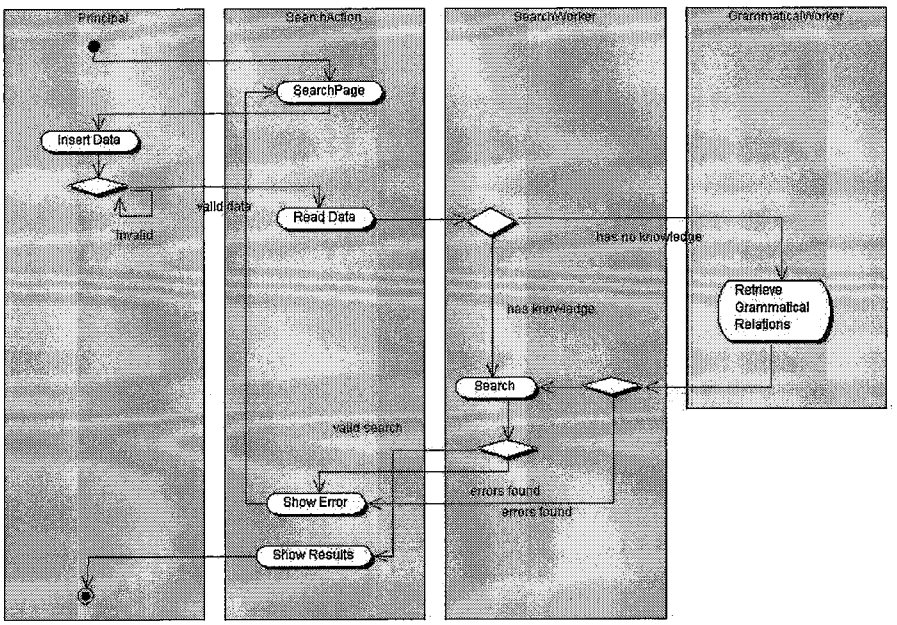

Figura 5.6: UC2.2 Pesquisa Automatizada - Diagrama de Actividade.

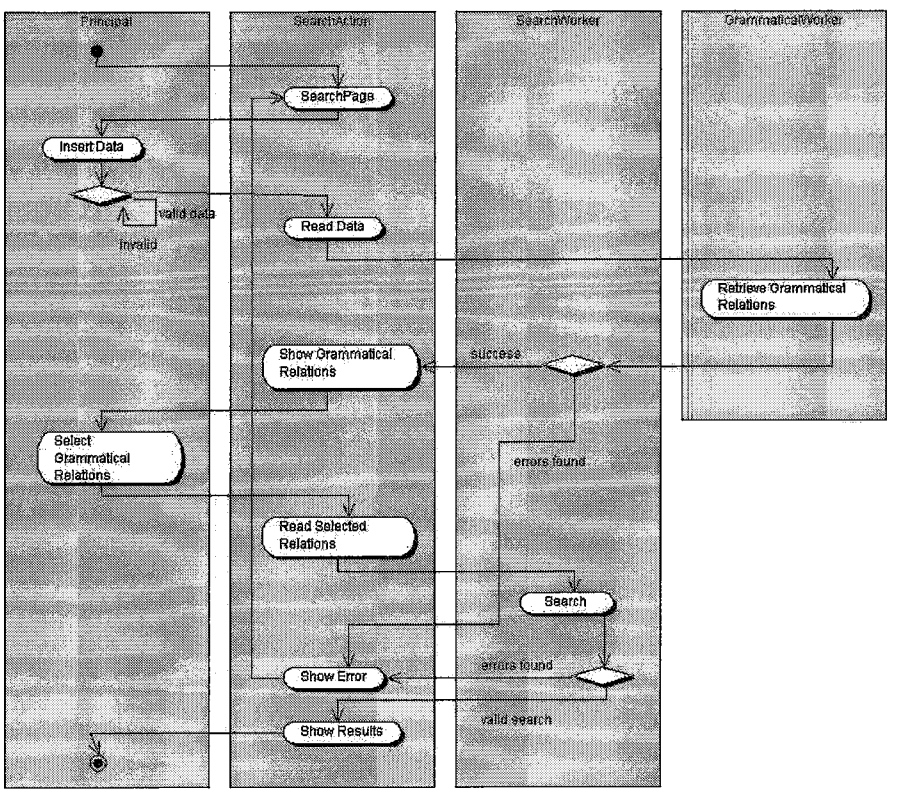

Figura 5.7: UC2.3 Pesquisa Interactiva - Diagrama de Actividade.

5.2.3 Sumarização

5.2.3.1 Descrição

Nome: UC3 Sumarização

Versão: 1.0

Autor: José Palmeiro

Última Actualização: 14/11/2005

Actores: Utilizador

Resumo: Esta funcionalidade permite que um utilizador autenticado sumarize documentos.

5.2.3.2 Fluxos

Inicialização: Este caso de uso é inicializado quando um utilizador aceda à página de sumarização documentos.

Sumarizar um Documento: O utilizador previamente autenticado pode aceder à funcionalidade de sumarização de documentos. Poderá posteriormente escolher qual o tipo de sumarização desejado, assim como a língua associada à sumarização. O diagrama de actividade descrito na figura 5.8 descreve o presente fluxo.

Finalização: Este caso de uso termina quando um for apresentado o sumário ao utilizador ou quando ocorram erros durante o processo de sumarização.

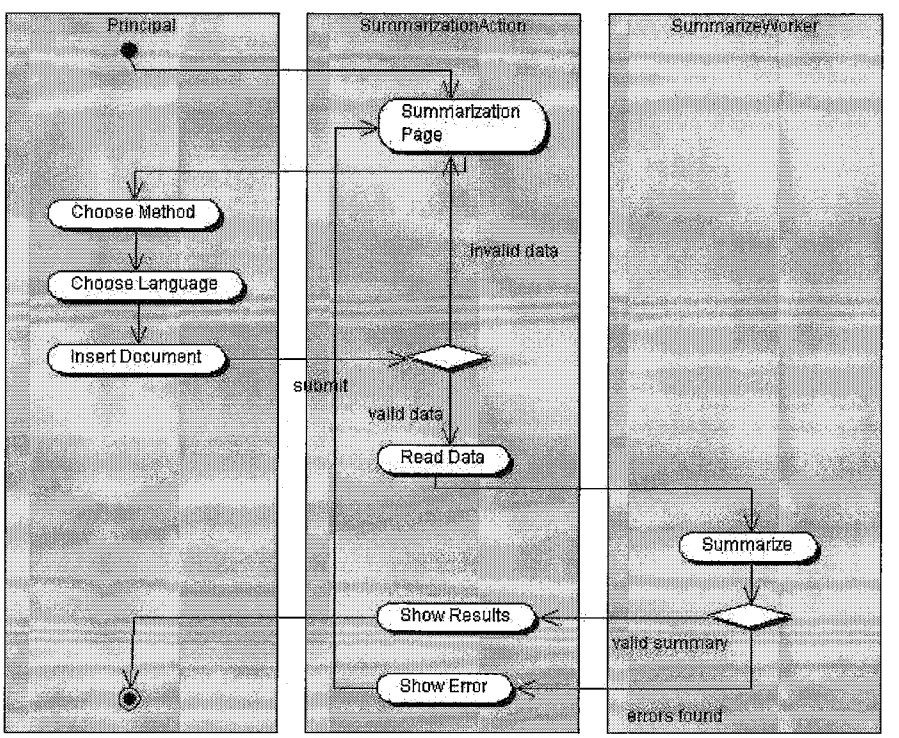

Figura 5.8: UC3 Sumarização de Documentos - Diagrama de Actividade.

5.2.4 Consulta de Mensagens

5.2.4.1 Descrição

Nome: UC4 Consulta de Mensagens

Versão: 1.0

Autor: José Palmeiro

Última Actualização: 11/02/2006

Actores: Utilizador

Resumo: Esta funcionalidade permite que um utilizador autenticado consulte mensagens provenientes de serviços e resultados de pesquisas antigas.

5.2.4.2 Fluxos

Inicialização: Este caso de uso é inicializado quando um utilizador aceder à página de consulta de mensagens.

Consultar Mensagens: O utilizador pode aceder à página de consulta de mensagens provenientes de serviços. Estas mensagens normalmente estão associadas a erros ocorridos e descritos pelos serviços disponíveis (p.e. pesquisa, sumarização), podendo também apresentar um carácter informativo sobre qualquer particularidade do sistema.

Para o caso em que o serviço de pesquisa não consiga responder atempadamente a um pedido realizado por um utilizador do sistema, este pode consultar posteriormente os resultados encontrados a partir deste fluxo. Esta funcionalidade é introduzida devido a dependência por serviços externos que respondem de forma assíncrona.

Finalização: Este caso de uso termina quando um for apresentada uma lista com os resultados encontrados.

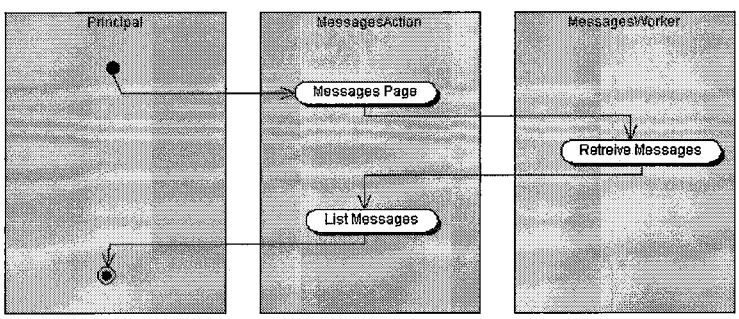

Figura 5.9: UC4 Consulta de Mensagens - Diagrama de Actividade.

5.2.5 Gestão do Conhecimento

Nome: UC5 Gestão do Conhecimento

Versão: 1.0

Autor: José Palmeiro

Última Actualização: 14/11/2005

Actores: Utilizador e Administrador

Resumo: Esta funcionalidade permite que um utilizador autenticado possa gerir as suas crenças e um administrador possa gerir todo o conhecimento associado a cada utilizador da plataforma.

5.2.5.1 Fluxos

Inicialização: Este caso de uso é inicializado quando um utilizador aceder a um dos fluxos abaixo descritos.

UC5.1 Gestão de Crenças: O utilizador pode interagir com o sistema adicionando ou removendo crenças adquiridas. O diagrama de actividade 5.10 exemplifica a adição de uma crença de forma manual, onde o utilizador selecciona o tipo de crença, o conceito base que deseja identificar, e uma associação que pode ser uma palavra ou um conjunto de palavras. A remoção de crenças é realizada através da página de listagem de crenças podendo o utilizador escolher as que deseja remover (ver diagrama de actividade 5.11).

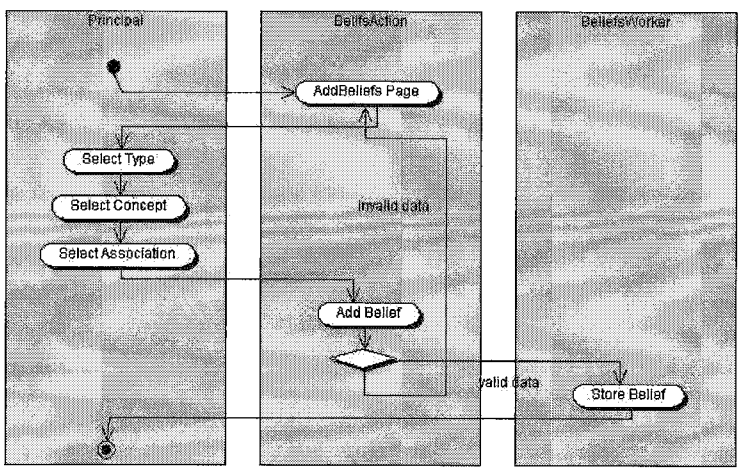

Figura 5.10: UC5.1.1 Adicionar Crenças - Diagrama de Actividade.

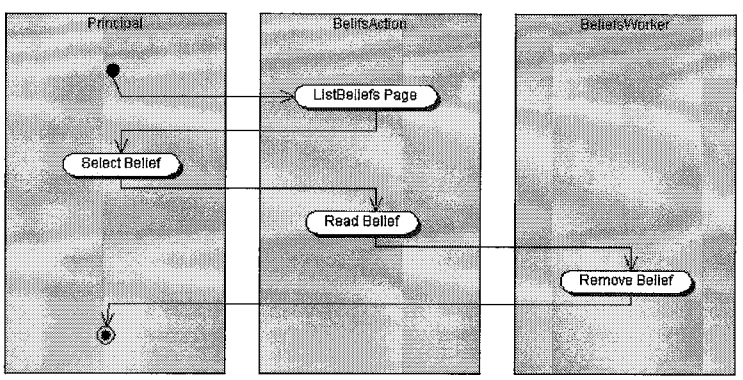

Figura 5.11: UC5.1.2 Remover Crenças - Diagrama de Actividade.

UC5.2 Gestão das Bases de Conhecimento: O administrador após de aceder à funcionalidade de gestão de bases de conhecimento é deparado com uma listagem de utilizadores do sistema. Através da selecção de um utilizador é carregada a respectiva base de conhecimento para visualização ou edição. Após a edição de uma base de conhecimento é possível armazená-la em memória persistente para que a próxima interacção com o utilizador possa reflectir as alterações realizadas (ver o diagrama de actividades 5.12). Com a presente funcionalidade o administrador poderá influenciar toda a unidade de memória lógica do sistema, podendo eventualmente adicionar, alterar, e remover regras de cada base de conhecimento de cada utilizador. Estas regras são as premissas

lógicas de actos e acções, sendo portanto a base principal para o seu desempenho. A sua edição requer um forte conhecimento sobre o tipo de lógica embebida no sistema, justificando-se assim o acesso exclusivo a um administrador.

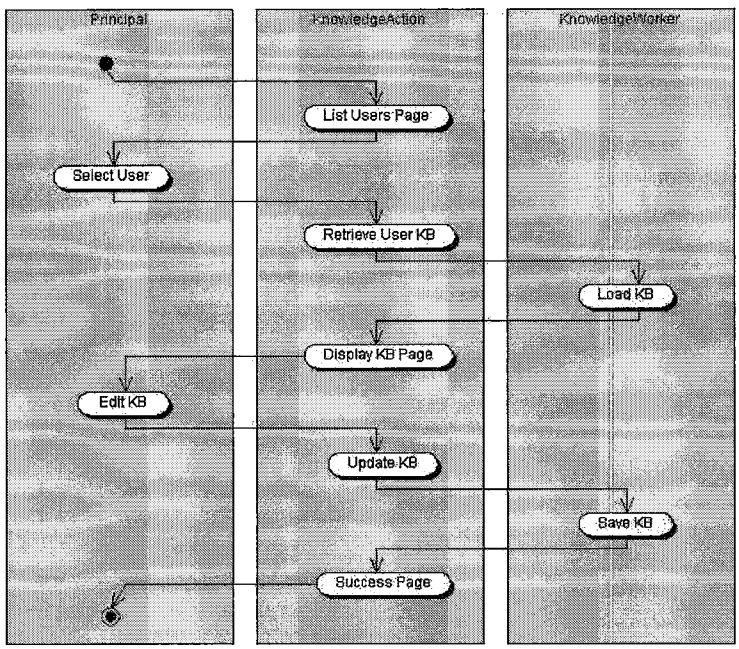

Figura 5.12: UC5.1.2 Gestão das Bases de Conhecimento - Diagrama de Actividade.

Finalização: Este caso de uso termina quando for adicionada ou removida uma crença (primeiro fluxo principal), ou actualizada ou consultada uma base de conhecimento (segundo fluxo principal).

5.3 Resumo

O presente capítulo fornece uma visão resumida do ponto de vista funcional, apresentando o modelo de análise funcional da plataforma Susy. Destacam-se cinco casos de uso principais e dois actores no sistema. Um utilizador uma vez autenticado, poderá interagir com o sistema de diversas formas, deste uma sumarização, pesquisas avançadas, consulta de mensagens, até à administração do sistema lógico, podendo eventualmente influenciar os resultados.

Este capítulo apresenta ainda as bases funcionais e essenciais, para que a arquitectura da solução proposta seja definida e apresentada no Capítulo 6.

Capítulo 6

Arquitectura de Software

O presente capítulo documenta a arquitectura da plataforma de *software* funcionalmente descrita no capítulo anterior.

6.1 Organização

A próxima secção descreve as decisões tomadas em função de requisitos não funcionais impostos à solução. As secções seguintes encontram-se organizadas em diferentes vistas sobre os modelos que, no seu conjunto, formam a especificação completa do sistema em *Unified Modeling Language* (UML):

- Vista Funcional - descreve os cenários significativos do sistema, evidenciando as entidades interessadas no sistema e que com ele comunicam;
- Vista Lógica - detalha os elementos e serviços que compõem o sistema, evidenciando a sua estrutura e a forma como interagem. Apresenta ainda as partes mais significativas dos modelos de análise e desenho;
- Vista de Execução - descreve os processos em ambiente de execução;
- Vista de Instalação - descreve a forma como os vários elementos do sistema (componentes, processos, entre outros) se relacionam com a infra-estrutura de *hardware*;
- Vista de Implementação - introduz regras de evolução e desenvolvimento, descrevendo ainda como está organizado o código no ambiente de desenvolvimento.

6.2 Decisões de Arquitectura

6.2.1 Objectivos

O objectivo desta arquitectura é apresentar uma plataforma de agentes que encapsule um modelo de negócio. Nesta plataforma deverão existir agentes que interagem com o utilizador através de uma aplicação *web*, enquanto outros deverão ser expostos numa arquitectura orientada a serviços. Para tal usou-se como plataforma base de desenvolvimento a *Java Platform, Enterprise Edition* (Java EE)[1] com tecnologia de agentes para a linguagem Java.

É ainda objectivo criar um modelo de separação claro entre as diversas camadas, através do padrão *Model-View-Controller* (MVC).

Estas tecnologias juntas formam as premissas da arquitectura proposta.

[1] Ver http://java.sun.com/javaee

6.2.2 Condicionantes

Como condicionantes da arquitectura encontram-se:

1. Fundações *web* codificadas na linguagem Java, através da *Java Platform, Standard Edition* (Java SE)[2] e Java EE;
2. Fundações de agentes codificadas através da *Java Agent DEvelopment Framework* (JADE)[3];
3. Representação de conhecimento modelado através paradigma DLP.

6.2.3 Estilo Arquitectural

A solução encontrada segue vários estilos arquitecturais, nomeadamente:

- Arquitectura de Agentes (MAS) e DLP;
- Arquitectura *Web*;
- Arquitectura Orientada a Serviços.

As fundações da arquitectura residem no dois primeiros estilos (ver figura 6.1), sendo o último considerado um complemento para a demonstração da potencialidade e evolução do sistema. A união destes estilo permite usufruir das melhores características de cada um.

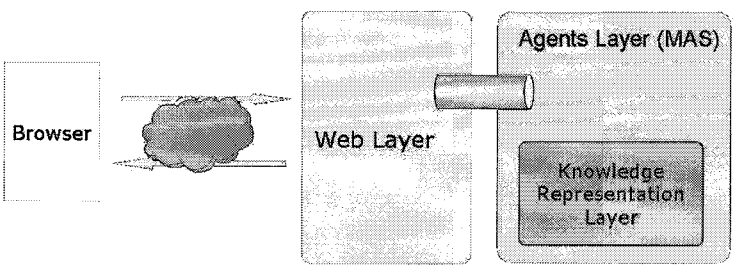

Figura 6.1: Plataforma Susy - MVC, MAS e DLP.

Para a arquitectura *web* será usada a aproximação *Model 2*, exemplificada na figura 6.2. Uma arquitectura *Model 2* introduz a noção de um controlador (através de *servlets*[4]) entre o *browser* e as páginas finais. O controlador centraliza a lógica de recepção de pedidos e de entrega de respostas baseado no *Uniform Resource Locator* (URL) de pedido, nos parâmetros de *input* e no estado da aplicação. Esta camada controladora também é responsável pela selecção das páginas a visualizar, o que separa as *Java Server Page* (JSP) e *servlets* em duas camadas distintas. As aplicações baseadas numa arquitectura *Model 2* apresentam um nível de manutenção reduzido e são mas fáceis de evoluir, pois não existem referências directas entre "vistas". A camada controladora fornece ainda um ponto único de controlo para segurança, *logging* e encapsulamento de dados de entrada de forma utilizável por um modelo *back-end* baseado em MVC (ver secção 6.4.2.1 *Model View Controller*). Por estas razões, este tipo de arquitectura é recomendada para a maior parte das aplicações interactivas, como é um caso de uma aplicação *web*.

[2]Ver http://java.sun.com/javase/
[3]Ver http://jade.tilab.com/
[4]Ver http://java.sun.com/products/servlet/.

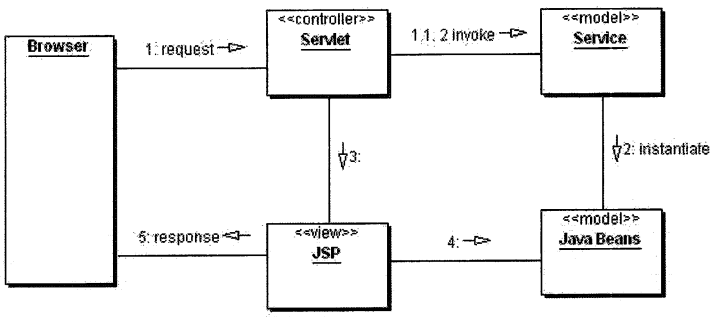

Figura 6.2: Arquitectura *Web Model* 2.

6.2.4 Tecnologias e Fundações de Arquitectura

6.2.4.1 Fundações *Web*

Para criar as fundações de arquitectura *web* foi escolhida a plataforma Java EE 1.4. Todas as bases serão construídas com esta linguagem, juntamente com a Java SE 5.

6.2.4.2 Agentes

Para criar a arquitectura de agentes, ou MAS, foi escolhida a plataforma JADE. Esta plataforma fornece a robustez necessária, não só pelo nível de desenvolvimento actual e implementação do standard FIPA, mas também pela larga comunidade existente que por si só representa uma garantia de qualidade.

A representação de conhecimento usada pelo MAS foi modelada através do paradigma DLP, com base na linguagem LUPS. Devido à limitada interface Prolog-Java[5], não só ao nível do "centralizado"[6] acesso via *Java Native Interface* (JNI), mas também pela lenta mas distribuída alternativa via *Transmission Control Protocol* (TCP), tornou-se tecnicamente inviável usar a implementação LUPS em XBS Prolog[7] numa plataforma com estas características. Como tal foi desenvolvido um componente Java que simula em boa parte o comportamento desta linguagem. O JLups[8], usando como núcleo o Mandarax[9], fornece uma forma escalável de representar o conhecimento numa infraestrutura complexa e distribuída.

6.2.4.3 Arquitectura Orientada a Serviços

A arquitectura orientada a serviços passará pela disponibilização de alguns agentes como *web services*. Para tal usaram-se *proxies* com tecnologia *Java API for XML Web Services* (JAX-WS)[10] - disponível na Java EE 5.

[5]Testado com a interface InterProlog (ver http://www.declarativa.com/interprolog), para uso da linguagem LUPS na forma original em XSB-Prolog.

[6]No sentido de não ser possível existir várias bases de conhecimento diferentes para cada UA. Com a opção JNI teria que existir uma base de conhecimento comum e partilhada por todos os UAs.

[7]Ver http://xsb.sourceforge.net.

[8]A arquitectura deste componente está descrita no Capítulo 7.

[9]Ver http://mandarax.sourceforge.net.

[10]Ver http://java.sun.com/webservices/jaxws.

6.3 Vista Funcional

6.3.1 Resumo do Sistema

O sistema desenvolvido é apresentado na forma de uma aplicação *web*, incidindo sobre uma plataforma orientada à recuperação de informação. Neste sistema existem dois perfis diferente de utilização, o utilizador normal e o administrador. Como tal, existem vários utilizadores que se podem autenticar e interagir com o sistema.

Como descrito no capítulo anterior, várias funcionalidades estão disponíveis (consultar secção 5.1), podendo o utilizador realizar pesquisas, sumarizar documentos, gerir as crenças assumidas pelo sistema, enquanto o administrador pode gerir e administrar toda a lógica embebida no sistema.

Parte das funcionalidades disponibilizadas são fornecidas por serviços, que estarão abertas a outros pontos de entrada no sistema.

6.3.2 Identificação de Serviços

Como parte inicial do processo de construção de software orientado a serviços destaca-se o processo de identificação de serviços (ver Capítulo 4, secção 4.2.2). De acordo com a informação apresentada no capítulo anterior, e com base nos diagramas de actividades aí introduzidos, é possível identificar os seguintes serviços através de uma abordagem *Top-Down*, Orientada a Processos de Negócio[11] com a técnica *operation-first* (ver secção 4.2.2.1):

- Serviço de Pesquisa, identificado[12] no caso de uso da secção 5.2.2 (consultar diagramas de activdade 5.5, 5.6 e 5.7);
- Serviço de Apoio Gramatical, identificado no caso de uso da secção 5.2.2, nas pesquisas automatizadas (ver diagrama 5.6) e interactivas (ver diagrama 5.7);
- Serviço de Sumarização de Documentos, identificado no caso de uso da secção 5.2.3 (consultar diagrama 5.8);

Cada serviço deverá funcionar de forma autónoma, gerindo o seu ciclo de vida e estando disponível a qualquer utilizador ou cliente do sistema.

6.3.3 Identificação de Agentes

Os serviços acima identificados serão fornecidos através de agentes, que poderão ter *façades* de *web services* para bem da interoperabilidade. Note-se que se os agentes já estivessem identificados na arquitectura, e fossem reutilizados nesta fase, teria sido usada a metodologia *Bottom-up* para exposição de agentes como *web services*, na secção anterior.

No entanto este conjunto por si só não representa um MAS. Existe uma necessidade natural de mais intervenientes, para bem da colaboração, estabilidade e evolução do sistema. Como tal são necessários mais agentes, nomeadamente de controlo, comunicação e pessoais.

Existem vários estudos [DSW97][KMA+01] sobre aproximações multi-agente em vários domínios para lidar com informação web. Na generalidade, uma arquitectura típica para este tipo de sistema é normalmente baseada num modelo de três camadas [CAC04]:

[11] Na forma de diagramas de actividade.

[12] Identificado como *Worker* nos diagramas de actividades.

- **Utilizador → Interacção com o Sistema.** Esta camada é responsável pela interacção com os utilizadores. Estes agentes (Agente do Utilizador ou Agentes de Interface) podem usar técnicas diferentes, tal como aprendizagem, comunicação entre utilizador e sistema.

- **Agentes do Utilizador → Agentes de Tarefas.** Camada que contém um conjunto de agentes especializados que com um objectivo muito bem definido. Normalmente são chamados de *Task Agents* embora sejam referidos por *Middle Agents*, *Execution Agents*, *Planning* ou *Learning Agents* noutras arquitecturas.

- **Agentes Intermédios → Agentes Web.** Esta camada engloba agentes que fornecem serviços, nomeadamente Agentes *Web*, *SoftBots*, *Crawlers*, entre outros que se especializam em recuperação e filtragem de informação da *web*, ou de outro repositório de dados.

A partir da análise realizada no capítulo anterior (ver Capítulo 5) é possível identificar os agentes responsáveis pelos fluxos e enquadrá-los numa estrutura de camadas análoga à descrita acima.

A Camada Frontal análoga à primeira anterior, vai conter todo um conjunto de agentes dedicados por exclusivo aos seus utilizadores. Cada um destes agentes - *User Agent* (UA) - manifesta as seguintes características:

- Aprendizagem e adaptabilidade, no sentido em que aprendem através da interacção com o utilizador e melhoram o seu comportamento;

- Autonomia, pois podem escolher vários fluxos de execução para satisfazer os pedidos do utilizador;

- Cooperação, devido ao diálogo constante com outros agentes, principalmente serviços;

- Pro-actividade, pois informam o utilizador de novos resultados e mensagens, podendo eventualmente detectar intenções sem estímulos externo do utilizador;

- Continuidade Temporal, preservando o conhecimento adquirido em interacções anteriores.

Esta camada contém ainda os agentes que residem no ambiente mais próximo ao cliente (por exemplo, numa *applet*, numa sessão *web*) que servem como canal de comunicação com o sistema, e com os respectivos agentes principais do utilizador - implementando o padrão de desenho *Proxy*, herdando assim o nome de *Proxy Agent* (PA). Este tipo de agente é mais simples e apenas apresenta indícios primários de comunicação, cooperação, e pura reactividade.

A camada intermédia é composta pelos agentes de controlo, igualmente análoga à camada dos *Task Agents*. Nesta camada residem agentes únicos em todo o sistema que o controlam e o mantêm coerente. Existe um agente que controla o ciclo de vida de todos os agentes que fornecem serviços - gentilmente nomeado de *Nanny Agent* (NA). Existe um outro agente que controla o lançamento dos agentes na camada frontal, sendo nomeado do *Controller Agent* (CA).

O NA manifesta as seguintes características:

- Autonomia, controlando o seu ciclo de vida;

- Comunicação, com todos os agentes de serviço;

- Reactividade, pois executa determinadas tarefas se detectar determinados estímulos, como a morte de um serviço.

O CA apresenta-se como um agente simples manifestando autonomia de desejo por criação.

A camada de serviços, tal como as anteriores herda os conceitos aplicados na última camada de [CAC04]. Aqui residem todos os agentes que fornecem serviços, destacando-se:

- O Agente de Pesquisas, ou *Search Agent* (SA);
- O Agente de Sumarização de Documentos, ou *Document Summarization Agent* (DSA);
- O Agente Gramatical, ou *Grammatical Agent* (GA).

Todos estes agentes mantêm um *hearbeat* interpretado pelo NA. A abordagem segue o modelo *yellow pages*, ou *directory agent systems*, em detrimento de um modelo "*brokered*". Com este tipo de sistema os "*requesters*" conhecem as capacidades do sistema, enquanto os "*providers*" desconhecem as preferências dos "*requesters*" [DSW97, p.1].

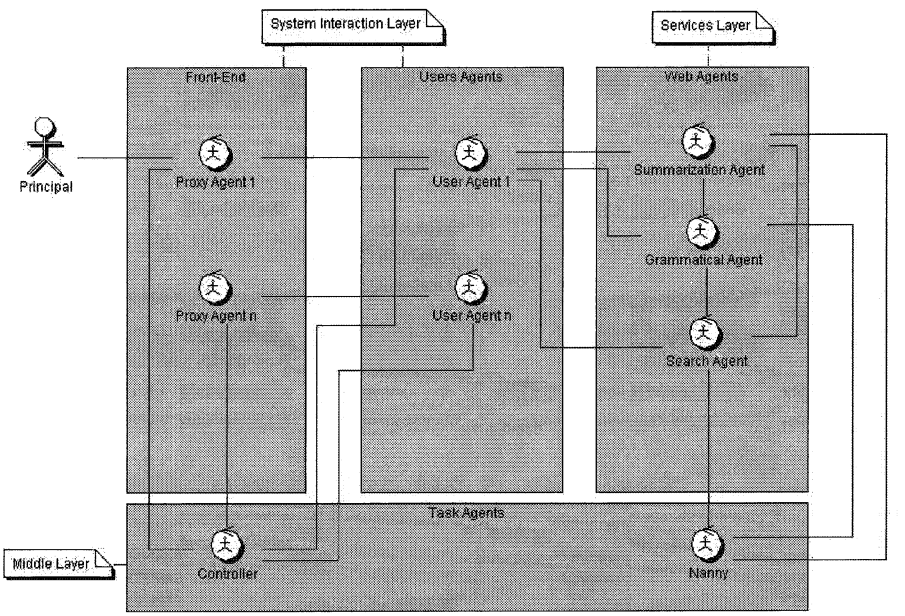

Figura 6.3: Arquitectura de Agentes.

Na presente arquitectura todos os agentes apresentam características bem definidas e adaptáveis à camada em que se inserem, sendo um sistema distribuído por natureza. Todas estas camadas podem estar distribuídas por vários contentores de agentes para uma possível distribuição funcional ou de carga.

Note-se ainda que a camada frontal onde residem os agentes mais próximos do utilizador, é a camada que poderá crescer em número de agentes pois por cada utilizador autenticado irão existir dois agentes. Todos os outros agentes das restantes camadas irão ser únicos em todo o sistema.

6.3.4 Casos mais Significativos para a Arquitectura

6.3.4.1 Sumarização de Documentos

Descrição: Processo de Sumarização de Documentos

Critérios de Selecção: Este fluxo é considerado significativo para a arquitectura pois descreve como é fornecido um serviço através de um agente e como é realizada a sumarização de documentos (qual o componente usado, e as opções disponíveis). Todos os restantes serviços seguem o mesmo desenho.

6.3.4.2 Agente do Utilizador, Ciclo de Vida

Descrição: Ciclo de Vida do Agente do Utilizador (UA)

Critérios de Selecção: Este fluxo é considerado significativo para a arquitectura pois descreve todo o processo de autenticação de um utilizador no sistema, podendo ser analisado em detalhe a inicialização dos agentes associados, assim como a finalização e limpeza do sistema aquando de um *logout*.

6.3.4.3 Agente do Utilizador, Gestão de Crenças e Intenções

Descrição: Gestão de Crenças e Intenções

Critérios de Selecção: Este fluxo é considerado significativo para a arquitectura pois descreve o núcleo de processamento do UA, desde a recepção de pedidos do utilizador até à sua realização. Este processo inclui a gestão crenças, detecção de intenções, e descreve a forma como o utilizador interage com a qualidade da informação e com o próprio sistema em si.

6.3.4.4 Pesquisa Interactiva

Descrição: Pesquisa Interactiva, Colaboração e Apoio à Recuperação de Informação

Critérios de Selecção: Este fluxo é considerado significativo para a arquitectura pois descreve um exemplo de cooperação entre agentes de serviços, juntamente com o agente do utilizador. Também explora o uso de tecnologias assíncronas como forma de auxiliar o retorno de informação.

6.4 Vista Lógica

Esta secção apresenta um resumo dos elementos mais significativos do desenho. Descreve também os mecanismos de arquitectura a serem utilizados no desenho da solução, bem como *frameworks* ou padrões que tenham sido seleccionadas para a realização da mesma.

6.4.1 Descrição Geral da Estrutura e Subsistemas

Para o presente projecto considerou-se a seguinte decomposição lógica do sistema em subsistemas, organizados de acordo com as camadas abaixo descritas e de forma a promover a reutilização. Cada subsistema poderá ainda ser decomposto acordo com as *tiers* relativas à sua distribuição no ambiente de execução.

6.4.1.1 Camada Específica da Aplicação

Contém os subsistemas usados numa aplicação ou projecto em particular, sendo os elementos menos reutilizáveis. Dentro desta descrição enquadram-se os módulos:

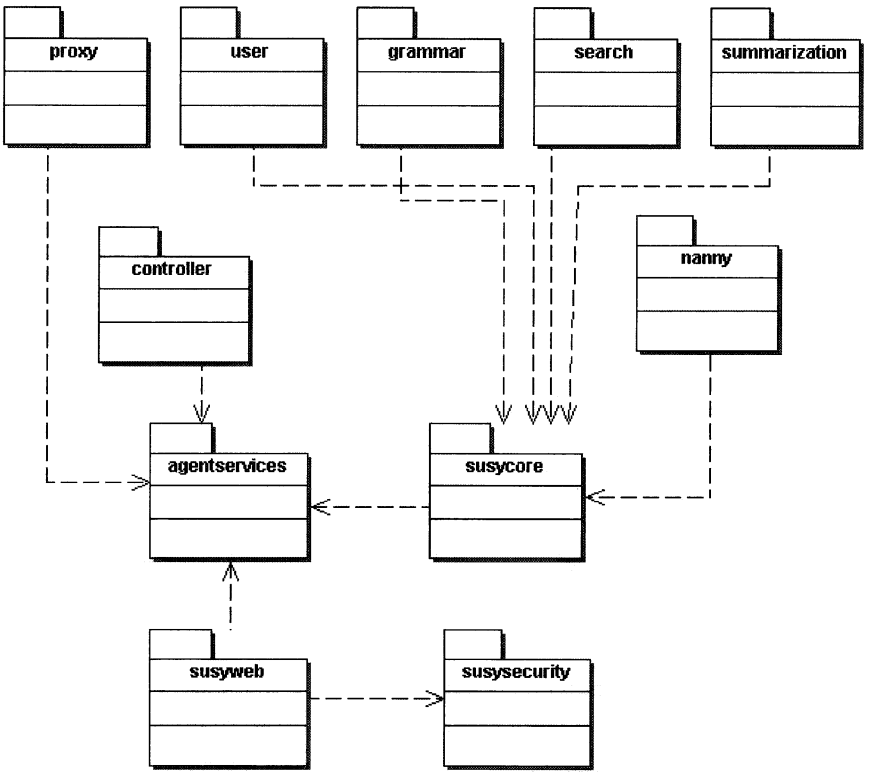

Figura 6.4: Decomposição Lógica do Sistema.

Applicação *Web*

Módulo que irá conter todo os componentes *web*, nomeadamente a camada *controller*, a *view*, assim como outros ficheiros de configuração.

Segurança

Contém o *realm* de autenticação para a aplicação *web*, através de JAAS (ver secção 6.4.4.1).

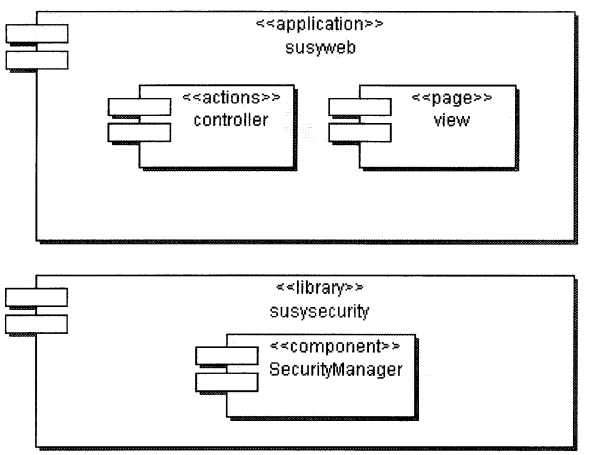

Figura 6.5: Camada Específica da Aplicação - Diagrama de Componentes.

6.4.1.2 Camada de Agentes e Serviços de Negócio (*Agents and Business-Services layer*)

Nesta camada enquadram-se todos os serviços e agentes disponíveis no sistema, nomeadamente:

Serviço de Consultas (SA)

O serviço de consultas é fornecido por um agente especializado em consultas *web*. Recebe mensagens que são pedidos de pesquisas e devolve uma listagem com resultados, links e descrições.

Serviço de Consultas Gramaticais (GA)

O serviço de consultas gramaticais é fornecido por um agente especializado em consultas gramaticais, recebendo mensagens como pedidos e devolvendo respostas representando relações gramaticais existentes com a expressão pedida.

Serviço de Sumarização de Documentos (DSA)

O serviço de sumarização de documentos é fornecido por um agente especializado em sumarizações, recebendo mensagens que numa primeira fase representam documentos textuais devolvendo uma frase representativa do documento e um sumário associado.

Agente de Apoio ao Utilizador (UA)

Agente especializado e totalmente dedicado a um utilizador, recebendo os seus pedidos e transformando-os em intenções a concretizar.

Serviço/Agente de Apoio à Comunicação (PA)

Agente que serve de canal de comunicação (*proxy*) entre a plataforma Java EE e o UA.

Agente de Controlo do Sistema (NA)

Agente que controla todos os serviços presentes nesta secção. Através do controlo do seu ciclo de vida, este agente poderá monitorizar todos os serviços presentes no sistema.

Agente de Inicialização de Cliente (CA)

Agente que é responsável de fornecer ao um cliente uma forma de ele interagir com o sistema, criando os agentes necessários para tal interacção (UA e PA).

6.4.1.3 Camada Específica ao Domínio de Negócio (*Business-Components layer*)

Esta camada contém subsistemas que se aplicam ao presente domínio mas são independentes da aplicação, podendo eventualmente ser reutilizados em aplicações no mesmo domínio ou noutro contexto diferente. Estes componentes estão acessíveis por interfaces e armazenam as respectivas implementações, sendo usados pelos serviços e agentes descritos na secção anterior:

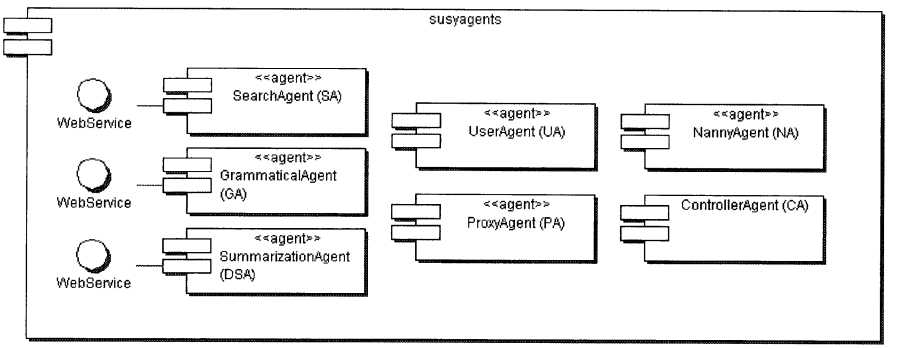

Figura 6.6: Camada de Serviços de Negócio - Diagrama de Componentes.

Componente de Pesquisas

O componente de pesquisas usa o motor de pesquisas Google[13] como implementação, sendo um cliente de um *web service* disponibilizado por esta organização.

Componente de Sumarização

O componente de sumarização usa como implementação o sumarizador de documentos Sue [Pal03]. Este sumarizador é puramente extractivo e fornece dois métodos para sumariar.

Componente Gramatical

O componente de consultas gramaticais usa como base da sua implementação o sistema de referências lexicais WordNet[14].

Módulo de Memória Lógica

O componente que fornece ao agente do utilizador todo o processo criativo, interpretando os pedidos do utilizador e transformando-os em intenções.

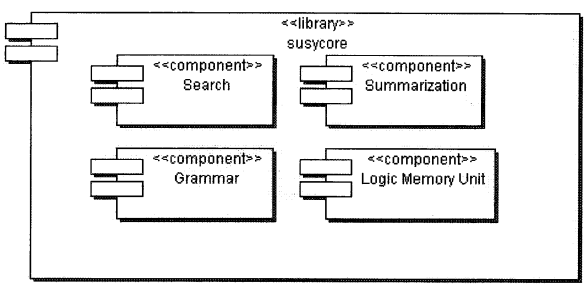

Figura 6.7: Camada Específica ao Domínio de Negócio - Diagrama de Componentes.

6.4.1.4 Camada Genérica (*Generic-layer / Application Framework*)

Esta camada possui uma vasta capacidade de reutilização uma vez que contém elementos aplicáveis a todos os componentes da camada específica ao domínio do negócio. Nela podemos encontrar a biblioteca AgentServices que fornece um conjunto alargado de funcionalidade, das quais se destacam (ver figura 6.8):

[13] http://www.google.pt
[14] http://wordnet.princeton.edu

Gestão de Agentes

Disponibiliza funcionalidades úteis para gestão do ciclo de vida de agentes, como o lançamento, registo no *Directory Facilitator*, entre outros.

Gestão da Plataforma de Agentes

Oculta a biblioteca de agentes usada, neste caso a plataforma JADE.

Gestão do Controller

Canal de comunicação entre a camada *controller* e a *model* (ver secção 6.4.2.1).

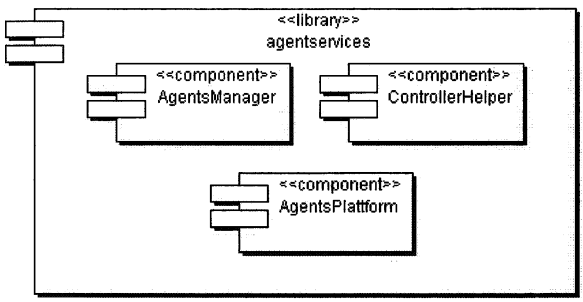

Figura 6.8: Camada Genérica - Diagrama de Componentes.

6.4.1.5 Organização Interna dos Subsistemas

Para cada subsistema acima introduzido considerou-se uma subdivisão em camadas relativos à sua distribuição no ambiente de execução. Os diferentes subsistemas estão divididos da seguinte forma (ver figura 6.9):

- Na *tier presentation* consideram-se os elementos relacionados com a interface com o utilizador, que realizem a formatação e conversão de dados de negócio a apresentar, ou que permitam interpretar quais as operações disponíveis e a forma das invocar;
- Na *tier agentservices* consideram-se todas as classes que representam o *middleware* de comunicação da camada de apresentação com a camada de negócio;
- Na *tier agent* consideram-se os agentes presentes no sistema, assim como os seus comportamentos;
- Na *tier business* consideram-se todos os serviços e entidades de negócio, bem como elementos auxiliares à execução das suas responsabilidades;
- Na *tier integration* consideram-se os elementos de ligação a mecanismos de memória persistente.

6.4.2 Padrões de Desenho

Nesta secção apresentam-se os padrões de desenho usados na arquitectura da solução.

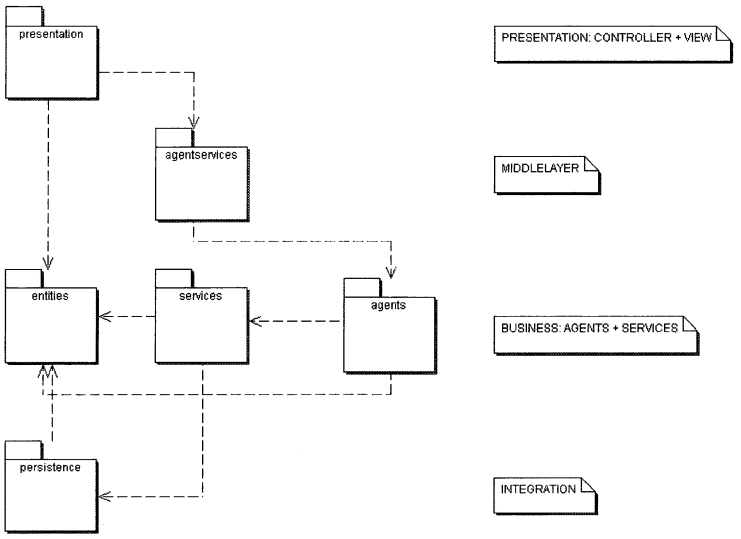

Figura 6.9: Organização Interna dos Subsistemas - Diagrama de Componentes.

6.4.2.1 Model View Controller

O padrão *Model-View-Controller* (MVC) é bastante conhecido e utilizado na construção de *software* actual. Foi criado como uma *framework* desenvolvida por Trygve Reenskaug para a plataforma Smalltalk em final dos anos 70. Desde então tem tido um papel bastante influente em *frameworks* gráficas e no seu desenho.

O *Model-View-Controller*, tal como o nome indica considera três perspectivas diferentes. A *Model* é uma perspectiva que representa informação de domínio, contendo todos os dados e comportamentos não usado na interface do utilizador.

A *View* representa a visualização da *Model* na interface do utilizador. Qualquer interacção realizada entre utilizador e o sistema é da responsabilidade do último interveniente de uma arquitectura *Model-View-Controller*, *Controller*. A camada *Controller* recebe o *input* do utilizador, manipula a *Model* e faz com que a *View* se actualize de forma apropriada.

Desta forma um modelo *Model-View-Controller* separa a camada de apresentação do negócio e o controlo do fluxo da aplicação da camada de apresentação. A primeira separação é uma heurística fundamental para o bom desenho de *software*, sendo de destacar várias razões que a justifiquem [Fow02]:

- Fundamentalmente a *model* e a *view* apresentam diferentes preocupações. Na *view* é importante desenhar uma boa interface e trabalhar com mecanismo gráficos. A camada *model* está focada no domínio de negócio, integração possivelmente com outros sistemas, entre outros. A probabilidade de se usarem bibliotecas diferentes para o desenvolvimento destas duas camadas é elevada, o que pode indiciar uma especialização de pessoal numa das áreas técnicas;

- Esta separação permite criar várias camadas de apresentação (p.e. para vários tipos de utilizadores) de forma independente do negócio (*model*). Inclusive podem ser usados vários tipos de clientes (uma interface *web*, uma *applet*, entre outros) para a mesma camada *model*;

- Testes em objectos não visuais são normalmente mais fáceis que em visuais. Com este modelo é possível construir uma bateria de testes sobre a camada de domínio de negócio;

É importante notar a direcção das dependências, pois enquanto a apresentação depende da *model* esta última camada não depende da primeira. Quem constrói a *model* não tem que saber qual a apresentação a ser usada. Esta solução simplifica a tarefa de desenvolvimento de ambas as camadas e torna mais fácil o processo de manutenção.

A segunda divisão separa a *view* da *controller* e permite um comportamento altamente dinâmico no fluxo da aplicação. A camada *controller* pode ser interpretada como uma *strategy* para a *view*, sendo esta potencialmente reutilizada para vários fluxos. A *view* apenas interpreta os valores recuperados da *model*, sendo o resultado final dum fluxo iniciado na *controller*.

A aplicação *web* irá recorrer à *framework open-source* Struts[15] e à biblioteca *Direct Web Remoting* (DWR)[16] como implementação do *design pattern Model-View-Controller*.

A utilização deste padrão na presente solução sofre uma ligeira derivação de estilo, devido à utilização de tecnologia de agentes numa arquitectura Java EE. Como é possível ver na figura 6.9, não existe comunicação directa da camada *web* (apresentação) para a camada de negócio. A arquitectura proposta, por ser em si um conjunto arquitectural diversificado exige algumas camadas intermédias para que a interligação de diversas tecnologias seja feita de forma harmoniosa. Para tal, desenvolveu-se uma camada intermédia de comunicação entre a camada *controller* e a *model*, o módulo AgentServices (Ver secção 6.4.6.3), de forma a que o uso da tecnologia de agentes seja transparente para a arquitectura *web*. A *view* neste caso recebe objectos proveniente de agentes (*model*), em vez de classes típicas de negócio (p.e. um *Enterprise Java Bean* (EJB)) - embora não o sabendo. As vantagens e desvantagens subjacente ao uso deste componente são analisadas na secção 6.4.6.3.

6.4.2.2 Proxy

O padrão *proxy* baseia-se no uso de objectos *proxy* durante a interacção com um objecto. Um objecto *proxy* actua como um substituto para o actual objecto. O uso deste tipo de objectos é comum em protocolos de interacção remota com objectos. Por exemplo, quando um objecto precisa de interagir com um objecto remoto através de uma rede, a forma mas interessante de encapsular e esconder o mecanismo de interacção é através da utilização de um objecto *proxy* que medeia a comunicação entre o objecto que realiza o pedido e o objecto remoto.

No presente sistema este padrão é usado como forma de encapsular a forma de comunicar com agentes, neste caso os objectos remotos. O cliente, uma *servlet* ou acção Struts, usa um objecto *proxy* para comunicar com os agentes remotos. Este objecto está encapsulado no componente AgentServices. Os detalhes da implementação estão descritos na secção 6.4.4.2.

6.4.2.3 Singleton

O padrão *singleton* foi desenhado para restringir a instanciação de uma classe para um objecto. É útil quando somente é necessário um objecto para coordenar determinadas acções no sistema. É um padrão largamente usado no presente sistema.

6.4.2.4 Factory Method

O *factory method* pertence ao conjunto dos *creationals patterns*. Este padrão ajuda a modelar uma interface para criação de objectos, que na altura de criação pode decidir

[15]Consultar http://struts.apache.org.

[16]Consultar http://getahead.ltd.uk/dwr/.

qual classe instanciar. Pode-se considerar um "padrão fábrica" pois é responsável pela "manufactura" de um objecto. Este padrão promove *loose coupling* através da eliminação dos *bindings* de classes específicas da aplicação no código.

6.4.2.5 Business Delegate

Um *business delegate* actua com uma abstracção de negócio, ocultando a implementação dos serviços de negócio (*managers*). O uso deste padrão reduz largamente o nível de acoplamento entre a camada de *presentation* e camada de *business.*

Na presente aplicação foram usado vários objectos deste tipo que implementa o padrão *Factory Method* (ver 6.4.2.4) e que ao mesmo tempo apresenta as funcionalidades de um *business delegate.*

6.4.2.6 Decorator

Um *decorator* adiciona a habilidade de alterar dinamicamente o comportamento de um determinado objecto. Este padrão é usado em casos onde se deseja alterar, modificar, ou expandir repetidamente o comportamento de um determinado objecto em *runtime.*

Este padrão foi usado na aplicação *web* na "decoração" de *java beans* recuperados das bases de conhecimento.

6.4.2.7 Strategy

O padrão *strategy* consiste no desacoplamento de um algoritmo do seu "hospedeiro", encapsulando-o em diferentes classes. De forma simples, um objecto e o seu comportamento são separados e inseridos em duas classes. Desta forma é possível alterar o algoritmo a qualquer altura, podendo inclusivé variar independentemente dos seus clientes.

Existem várias vantagens na utilização deste padrão. Por exemplo, se existem vários comportamentos que um objecto deve realizar, é mais fácil geri-los e mantê-los em diferentes classes do que no corpo de um método dessa mesma classe. A tarefa de alterar, remover ou adicionar comportamentos torna-se fácil pois cada um está numa classe separada.

Este padrão permite construir software como uma colecção *loosely coupled* de parte ou componentes que se podem substituir uns aos outros, em contraste com um sistema monolítico *tightly coupled. Software loose coupled* apresenta-se mais extensível, evolutivo, com manutenção mais fácil e maior reutilização.

Como exemplo deste padrão no presente projecto destacam-se todos os algoritmos do componente JLups, que lêem e gravam as bases de conhecimento. Estes algoritmos estão todos em classes separadas embebidos numa classes que os escolhe e usa, portanto num padrão *strategy.*

6.4.2.8 Façade

Estruturar o sistema em subsistemas ajuda a reduzir a sua complexidade. Um objectivo comum de quem desenha é minimizar a comunicação e as dependências entre subsistemas. Uma forma de satisfazer este objectivo é fornecer um objecto tipo façade, uma interface simples e única para várias funcionalidades de um determinado subsistema.

6.4.3 Bibliotecas, Componentes e Frameworks

Todas as bibliotecas usadas na presente arquitectura são *open source*, assim como todo o *software* desenvolvido no presente documento.

6.4.3.1 Struts

Foi usada a *framework* Struts[17] para implementar o modelo MVC (ver secção 6.4.2.1).

6.4.3.2 JADE

A plataforma JADE[18] (*Java Agent DEvelopment Framework*) é uma *framework* de agentes totalmente implementada na linguagem Java. Simplifica a implementação de um MAS através de *middleware* que respeita as especificações FIPA a através de uma ferramenta gráfica que suporta *debugging* e *deployment*. A plataforma de agente pode estar distribuída por diversas máquinas (inclusive em sistemas de operação diferentes) e configurável através de uma GUI remota.

Todos os agentes presentes no sistema são agentes JADE.

6.4.3.3 Mandarax

O Mandarax[19] é uma biblioteca *open source* para dedução de regras, fornecendo uma estrutura para definir, gerir e realizar inferências sobre bases de conhecimento. É ainda uma implementação Java pura de um motor de regras que suporta múltiplos tipos de factos e regras baseadas em reflexão, bases de dados, EJB, entre outros. Também fornece um motor de inferência Java EE[20] *compliant* usando *backward chaining*.

Esta biblioteca foi usada como forma de implementar o mecanismo DLP (ver Capítulo 7).

6.4.3.4 AjaxTags

A biblioteca AjaxTags[21] é um conjunto de *tags* que simplificam o uso de tecnologia *Asynchronous Javascript And XML* (AJAX) nas JSPs. A sua integração com outras tecnologias (p.e. Struts) é bastante harmoniosa.

Esta biblioteca foi usada para tornar as paginações *web* assíncronas.

6.4.3.5 Direct Web Remoting

A *framework Direct Web Remoting* (DWR)[22], introduz a tecnologia AJAX com o objectivo de reduzir o tempo de desenvolvimento e a probabilidade de ocorrência de erros, através do fornecimento de funções bem conhecidas e da remoção da maior parte do código repetitivo, normalmente associado com *sites* altamente interactivos. Tal como a biblioteca descrita no ponto anterior, a sua integração com outras tecnologias é bastante harmoniosa.

Todo o enriquecimento AJAX foi desenvolvido com esta biblioteca.

[17] Consultar http://struts.apache.org/.
[18] Consultar http://jade.tilab.com/
[19] Consultar http://mandarax.sourceforge.net/
[20] Consultar http://java.sun.com/j2ee//
[21] Consultar http://ajaxtags.sourceforge.net/
[22] Consultar http://getahead.ltd.uk/dwr/index

6.4.3.6 DisplayTag

A *DisplaTag*[23] é uma biblioteca *open source* de *custom tags* que fornece padrões de apresentação *web* de alto nível integráveis num modelo MVC.

Esta biblioteca foi usada para paginação de resultados na camada de apresentação.

6.4.3.7 JLups

O motor de regras DLP é implementado através do componente JLups. Consultar Capítulo 7 para detalhes arquitecturais.

6.4.3.8 Sue

O *Sue* é um sumarizador extractivo de documentos desenvolvido na linguagem Java. Este componente foi desenvolvido no âmbito da cadeira Pesquisa de Informação em Bases de Texto, do Mestrado em Engenharia Informática. Mais informações consultar [Pal03].

Esta biblioteca é usada para a sumarização de documentos.

6.4.3.9 Google API

Foi usada a biblioteca[24] fornecida pela Google de acesso ao motor de busca através da linguagem Java.

Todas as pesquisas *web* são realizadas através desta biblioteca.

6.4.3.10 JWordNet

Foi usada a API Java[25] para acesso ao sistema de referências lexicais WordNet[26].

Esta biblioteca é usada para recuperar relações gramaticais.

6.4.3.11 Jakarta Commons

O projecto *Commons*[27] é um subprojecto Jakarta focado na reutilização de componentes Java.

Foram usadas várias bibliotecas do projecto *Commons*, nomeadamente *Validator* e *Beanutils*. Outras, como *Collections*, *Dbcp*, *Digester*, *Lang* e *Logging* foram também usadas de forma indirecta por outras *frameworks*.

6.4.3.12 JavaServer Pages Standard Tag Library

A *JavaServer Pages Standard Tag Library* (JSTL)[28] encapsula em forma de *tags* as funcionalidades comuns a várias aplicações *web*. Suporta processamento iterativo, condicional, manipulação de XML, internacionalização e uso da *Structured Query Language* (SQL). Suporta ainda integração com *custom tags*.

Foi usada a biblioteca JSTL em toda a camada de apresentação e como complemento às tags Struts.

[23] Consultar http://displaytag.sourceforge.net
[24] Consultar http://www.google.com/apis/
[25] Consultar http://jwn.sourceforge.net/
[26] Consultar http://wordnet.princeton.edu/
[27] Consultar http://jakarta.apache.org/commons/
[28] Consultar http://java.sun.com/products/jsp/jstl/

6.4.3.13 Java Authentication and Authorization Service

A *Java Authentication and Authorization Service* (JAAS)[29] é um conjunto de APIs que permite a autenticação e controlo de acesso. Implementa uma versão do *standard Pluggable Authentication Module* (PAM), e suporta autorização baseada em utilizadores.

Esta *framework* foi usada para implementar o mecanismo de autenticação.

6.4.4 Mecanismos

Nesta secção apresentam-se os mecanismos mais significativos usados na arquitectura da solução.

6.4.4.1 Autenticação

Descrição: A autenticação de utilizadores na plataforma Susy é realizada através da *framework Java Authentication and Authorization Service* (JAAS)[30]. Esta API encaixa numa arquitectura Java EE de forma independente de todo o código da aplicação, podendo para o efeito serem usados vários tipos de bases de utilizadores, nomeadamente um *Lightweight Directory Access Protocol* (LDAP), uma base de dados acessível via *Java Database Connectivity* (JDBC), entre outros. Neste caso por uma questão simplicidade é usado um ficheiro de texto com utilizadores, palavras chaves, e respectivos perfis. Não obstante, a alteração desta base de autenticação para uma outra mais evoluída é trivial e totalmente configurada fora das fronteiras da aplicação. O módulo de JAAS usado é o FileLogin[31]

Interface Pública:

Guia de Programação: A entrada no ficheiro de configuração de JAAS e realizada da seguinte forma:

```
FileLogin
{
  com.tagish.auth.FileLogin required debug=true pwdFile="/path/to/passwd";
};
```

O ficheiro de configuração que contém os utilizadores, palavras chaves e respectivos perfis apresenta o seguinte formato:

```
# Passwords for com.tagish.auth.FileLogin
test1:5a105e8b9d40e1329780d62ea2265d8a:administrator:user
palmeiro:c1a339f5034d7a27c9478dcdcea0e15d:user
jpalmeiro:c1a339f5034d7a27c9478dcdcea0e15d:user
jp:c1a339f5034d7a27c9478dcdcea0e15d:user
god:c1a339f5034d7a27c9478dcdcea0e15d:user
test2:ad0234829205b9033196ba818f7a872b
```

6.4.4.2 Interacção Utilizador-Agente / *Controller-Model*

Descrição: Mecanismo de comunicação entre Utilizador, Sistema e Agente Pessoal. Este componente encapsula a forma como é realizada a comunicação entre o utilizador, a camada *Controller*, e a camada *Model* onde residem os agentes.

[29] Consultar http://java.sun.com/products/jaas/

[30] Consultar http://java.sun.com/products/jaas/

[31] Disponível em http://free.tagish.net/jaas/.

Interface Pública: Consultar secção 6.4.6.3 Camada Genérica, Módulo AgentServices.

Guia de Programação: Por cada utilizador e respectiva sessão existe um objecto *ControllerHelper*. Através deste objecto as acções (*servlets*) podem enviar mensagens para o agente associado ao utilizador. Para tal, devem instanciar uma classe que implemente a interface *UserRequest* de forma a iniciarem um acto de fala com o agente, este também único por utilizador. Este objecto é composto por diversas características, nomeadamente se o acto de fala exige resposta (*request-reply*), qual o tipo de pedido (um identificador unívoco bem conhecido pelo sistema), o pedido em causa num formato de texto, e métodos para recuperar ou inserir crenças (usados na gestão manual de crenças - ver secção 6.4.7.3).

6.4.4.3 Gestão do Ciclo de Vida dos Serviços

Descrição: Controlo do *heartbeat* dos agentes da camada de serviço pelo NA, fornecendo um serviço de controlo e estabilidade de todo o sistema.

Interface Pública: Não aplicável.

Guia de Programação: Todos os detalhes da utilização e *internals* deste mecanismo encontram-se descritos na secção 6.5.

6.4.4.4 Validações

Descrição: As validações introduzidas na camada *view* estão associadas a campos de introdução de dados. Usa-se a API commons-validator juntamente com a *framework* de MVC Struts.

Na camada *model* são usadas *templates* para um agente "filtrar" mensagens.

Interface Pública: Para a camada *view* consultar documentação disponível no site[32].

Para a camada *model* consultar os *javadocs*[33] da plataforma JADE.

Guia de Programação: Exemplo da um ficheiro de definição de validações para a aplicação Susy. Cada campo de cada form está associado a determinado tipo de validações, e a uma determinada mensagem de erro. As mensagens de erro (p.e. susy.search.form.search) são definidas num ficheiro de propriedades, enquanto os identificadores de validações (p.e. *required*) são referências a métodos que se responsabilizam pela validação em causa, de forma genérica.

```
<form-validation>
  <global>
    <constant>
      <constant-name>date.format</constant-name>
      <constant-value>yyyyMMdd</constant-value>
    </constant>
    <constant>
     <constant-name>unsafeChars</constant-name>
        <constant-value>'|*|&amp;| |</constant-value>
    </constant>
```

[32] http://jakarta.apache.org/commons/validator/

[33] http://jade.tilab.com/doc/api/jade/lang/acl/MessageTemplate.html

```
  </global>
  <formset>
    <form name="SearchForm">
      <field property="message" depends="required">
        <arg0 key="susy.search.form.search"/>
      </field>
    </form>
    <form name="SummarizationForm">
      <field property="method" depends="required">
        <arg0 key="susy.form.summarization.summarization.method"/>
      </field>
      <field property="language" depends="required">
        <arg0 key="susy.form.summarization.summarization.language"/>
      </field>
      <field property="document" depends="required">
        <arg0 key="susy.form.summarization.summarization.document"/>
      </field>
    </form>
    <form name="BeliefForm">
      <field property="type" depends="required">
        <arg0 key="susy.beliefs.type"/>
        <var>
          <var-name>unsafeChars</var-name>
          <var-value>${unsafeChars}</var-value>
        </var>
      </field>
      <field property="base" depends="required">
        <arg0 key="susy.beliefs.base"/>
        <var>
          <var-name>unsafeChars</var-name>
          <var-value>${unsafeChars}</var-value>
        </var>
      </field>
      <field property="association" depends="required">
        <arg0 key="susy.beliefs.belief"/>
        <var>
          <var-name>unsafeChars</var-name>
          <var-value>${unsafeChars}</var-value>
        </var>
      </field>
    </form>
  </formset>
</form-validation>
```

6.4.4.5 JLups - Motor de Regras

Descrição: O JLups é um motor de regras Java orientado para actualizações dinâmicas e inspirado na LUPS.

Interface Pública: Consultar Capítulo 7.

Guia de Programação: Consultar Capítulo 7.

6.4.4.6 Registo de Operações

Descrição: O mecanismo de *logging* usado é o componente *java.util.Logger*. Através deste componente são reportados eventos com vários níveis de importância, podendo ser realizada uma filtragem por níveis.

Interface Pública: Consultar javadocs da Java SE 5[34].

Guia de Programação: Consultar javadocs da Java SE 5.

6.4.4.7 Paginação

Descrição: Foi usada a *tag library* DisplayTag[35] como mecanismo de paginação de resultados nas páginas *web*. Esta biblioteca *open source* permite uma integração harmoniosa de apresentação de resultados num modelo MVC.

Interface Pública: Ver Guia de Programação.

Guia de Programação: Abaixo segue uma extracção de código da página de listagem de crenças como exemplo da utilização desta forma de paginação no presente projecto. De forma resumida, esta *taglib* recupera uma lista de *java beans* armazenado num escopo (neste caso de *request*) e por reflexão (atributo *property* na tag display:column) apresenta os valores das propriedades destes *beans* em colunas. Cada *bean* corresponderá a uma nova linha a listagem. Todo o trabalho de ordenação, exportação de resultados são realizadas de forma interna à *taglib* não sendo evidenciadas na forma de utilização.

```
<display:table id="entry"
export="true" name="<%= Constants.RequestAttributes.RA_BELIEFS%>"
requestURI="/user/beliefs/show.do"
defaultsort="2" defaultorder="ascending" pagesize="10">
  <display:column sortable="true" titleKey="susy.beliefs.type"
   property="type"/>
  <display:column sortable="true" titleKey="susy.beliefs.base"
   property="base"/>
  <display:column sortable="true" titleKey="susy.beliefs.belief"
   property="belief"/>
  <display:column titleKey="susy.beliefs.remove">
    <a href="<c:url value="/user/beliefs/remove.do">
      <c:param name="rp_beliefType">
        <c:out value="${entry.type}"/>
      </c:param>
      <c:param name="rp_beliefBase">
        <c:out value="${entry.base}"/>
      </c:param>
      <c:param name="rp_beliefAssociation">
        <c:out value="${entry.belief}"/>
      </c:param>
    </c:url>">remove
    </a>
  </display:
```

6.4.5 Regras de Desenho

Nesta secção apresenta-se as regras de desenho que foram - e que devem ser - seguidas na evolução da presente plataforma.

[34] http://java.sun.com/JSE/1.5/docs/api/index.html
[35] http://displaytag.sourceforge.net

6.4.5.1 Acções Struts

Todas as classes responsáveis pelo tratamento da interacção dos utilizadores com o sistema, serão extensões à classe *AbstractBaseAction* que por sua vez é uma subclasse da *DispatchAction*, disponibilizada pela *framework* Struts.

Estas classes ficarão declaradas no ficheiro *struts-config.xml*, sendo declaradas para todas elas um atributo '*parameter*' com o valor '*op*'. Este parâmetro especifica o nome do parâmetro que cada uma destas classes espera receber para identificar a acção específica a ser executada, correspondendo sempre ao nome de um método com uma assinatura bem conhecida como de seguida se exemplifica:

Para o *link* correspondente ao URL "/user/searchAction.do?op=simpleSearch" ser interpretado deverá ser declarado no ficheiro *struts-config.xml* uma entrada como:

```
<action
 input="page.search"
 name="SearchForm"
 parameter="op"
 path="/user/searchAction"
 type="com.palmeiro...SearchAction"
 unknown="false"
 validate="false">
```

Deve-se ainda mascarar este *link* através de uma *forward action*, de forma a ocultarem-se os parâmetros:

```
<action
  forward="/user/searchAction.do?op=simpleSearch"
  input="page.search"
  name="SearchForm"
  path="/user/simpleSearch" />
```

O *link* final a ser usado nas páginas seria "/user/simpleSearch.do".

A classe correspondente (*SearchAction*) deverá ser declarada da seguinte forma:

```
public class SearchAction extends AbstractBaseAction
```

Neste caso deve existir na classe a declaração de um método com a seguinte assinatura:

```
public ActionForward simpleSearch(ActionMapping mapping,
                                  ActionForm form,
                                  HttpServletRequest request,
                                  HttpServletResponse response);
```

De forma a reutilizar mecanismos comuns disponibilizados por estas classes, no âmbito do presente projecto será definida a classe abstracta AbstractBaseAction[36] que deverá ser estendida por todas as classes de interacção com o utilizador, que usem a *framework* Struts (Actions).

Deve considerar-se como *default scope* da aplicação o escopo de *request*, a menos que seja indicado especificamente um outro escopo deve-se considerar que todos os parâmetros enviados das classes de interacção para as páginas de visualização de conteúdos e vice-versa existem em *request* e deixarão de estar disponíveis no final do processamento do pedido que lhes deu origem.

[36] *Package com.palmeiro.susy.web.presentation.controller.AbstractBaseAction.*

6.4.5.2 Agentes de Serviços

Todos os agentes de serviços do sistema deverão estar registados na aplicação, assim com no *directory facilitator*. À parte do seu comportamento principal, deverão incluir também o comportamento *HeartbeatBehaviour* de forma a que o seu ciclo de vida seja controlado pelo NA.

6.4.5.3 População de Valores em *Forms Web*

O carregamento de valores pré-definidos em páginas *web* (p.e. *comboboxes*, entre outros) deverá ser realizado de forma assíncrona através da biblioteca DWR. Para tal as classes responsáveis pelo fornecimento destes valores deverão ficar acessíveis assincronamente através da sua configuração no ficheiro *dwr.xml*.

6.4.5.4 Logging

Devem ser geradas mensagens de *log* sempre que ocorra um caso de excepção no funcionamento da aplicação. Pretende-se que seja guardado o maior detalhe possível acerca da origem da excepção, portanto devem ser preservados os *stack traces* e as mensagens de erro do subsistema onde ocorre a excepção.

6.4.5.5 Tiles

De forma a simplificar a definição das páginas a apresentar ao utilizador, para cálculo das opções a mostrar no menu, foi usada a extensão Tiles para a *framework* de Struts. Todas as páginas *web* que compõem a camada *view* são compostas por quatro secções (ou tiles): *header*, *footer*, *menu* e *content*, sendo esta última a única alterada de fluxo para fluxo.

6.4.5.6 Paginação de Listas de Resultados

Todas as tabelas de valores a serem representadas na interface com o utilizador, devem ser definidas recorrendo à *taglib* identificada pela versão 1 do projecto *open source* DisplayTag[37].

6.4.5.7 Excepções

Os erros na aplicação *web* devem ser todos tratados pela *tier* de *presentation*, devendo as outras *tiers* garantir que todos os erros ocorridos são correctamente tratados e passados à *presentation*. Nesta, a acção inicial do utilizador deve ser redireccionada para uma página de erro com uma mensagem significativa sobre as circunstâncias em que o erro terá ocorrido. Aquando deste evento deverá ainda ser feito um log descrevendo a situação ocorrida.

O tratamento de excepções nas acções é realizado através do método *saveErrorAndLog*, disponível na classe *AbstractBaseAction*. Este método oculta a forma como o mecanismo de *logging* está implementado.

Todas as excepções que forem relançadas para o método que efectua a chamada devem adicionar à nova excepção, como argumento do construtor, uma referência para a excepção original.

[37]Para mais informações sobre a instalação, configuração e utilização desta framework deve ser consultado o site http://displaytag.sourceforge.net.

A *tier* de *business* deve tratar todas as excepções dos subsistemas que gere, passando-as para a camada de *business* como instância de *BusinessException* ou de uma das suas subclasses específicas. É também responsável pelo tratamento das excepções ocorridas na *tier* de *persistence*.

A *tier* de *persistence* deve tratar todas as excepções dos subsistemas que gere, passando-as para a camada de *business* como instância de *PersistenceException* ou de uma das suas subclasses.

6.4.5.8 Crenças e Intenções

Todas as acções que o utilizador possa desencadear, que utilizem as capacidades lógicas do seu agente pessoal, devem seguir as seguintes regras. Para adicionar comportamentos que lidem com crenças e intenções, será necessário:

1. Introduzir um novo mapeamento bem conhecido entre a interface *web* e a lógica interna do seu agente, de forma a que o agente perceba qual foi a atitude tomada pela utilizador;
2. Codificaçar de novas regras associadas aos novos comportamentos;
3. Possivelmente implementar um método que se responsabilize por um conjunto de operações a desencadear, de acordo com o resultado de uma regra;

São exemplo destas acções todo os tipos de pesquisas disponíveis. Mais detalhe consultar a secção 6.4.7.4 Pesquisa Interactiva, Colaboração e Apoio à Recuperação de Informação.

6.4.6 Elementos Mais Significativos do Desenho

Nesta secção apresentam-se os elementos mais significativos das diversas camadas que compõem esta plataforma.

6.4.6.1 Camada de Serviços de Negócio

A camada de serviços de negócio é fornecida através de agentes e *web services*. Um serviço ou agente é descrito para exemplificar as operações existentes e a forma como são processadas as mensagens.

Agente de Sumarização

Descrição: O agente de sumarização disponibiliza uma operações para sumariar documento. Esta operação corresponde a uma operação no WSDL do presente *web service*, assim como um tipo específico de uma mensagem ACL para o agente em causa. Cada agente responsável por um determinado serviço tem uma determinada *façade*, implementada por uma classe Java que o usa e elevada a *web service*.

Especificação Estrutural: Ver figura 6.10.

Especificação Comportamental: O diagrama de actividades (ver figura 6.11) exemplifica o comportamento do agente de sumarização de documentos. O fluxo é iniciado através do envio de uma mensagem ACL com a *performative REQUEST*, por parte de um cliente - normalmente um agente. Esta mensagem ACL contém uma estrutura XML com o documento em causa, o tipo de sumarização e a respectiva língua pretendida para

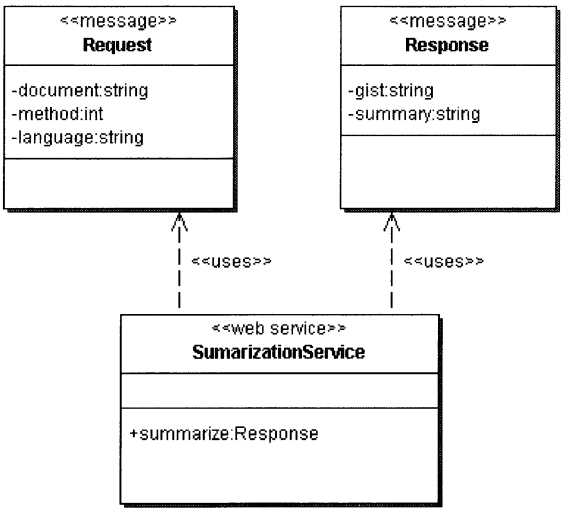

Figura 6.10: Interface da *façade* para o DSA - Diagrama de Classes.

o processo. Após a recepção desta mensagem, o DSA valida-a através de uma *template* específica. Caso não preencha os requisitos codificados na *template* de validação (por exemplo não ter a *performative REQUEST*) a mensagem é ignorada. Se a mensagem estiver bem formatada o fluxo prossegue e é invocada a sumarização disponibilizada pelo Sumarizador Sue [Pal03]. Após criado o sumário, é construída uma estrutura simples XML para conter o sumário e o *gist* do documento em causa. Este XML será embebido numa mensagem ACL que viajará até ao cliente como resposta. A realização deste serviço encontra-se detalhada - na forma de diagramas de sequência - no ponto 6.4.7.1 da secção 6.4.7 Realização de Funcionalidades mais Significativas.

6.4.6.2 Camada Específica ao Domínio

Núcleo - Módulo SusyCore

Descrição: Contém todo o núcleo e estrutura base da camada de negócio.

Classes Participantes: Este módulo disponibiliza os seguintes serviços (ver figura 6.12):

- GrammarManager: Interface do serviço gramatical, fornecendo métodos para recuperar sinónimos, hipónimos, hiperónimos, definições e outras informações relativas a uma palavra. Serviço usado directamente pelo agente gramatical (GA);

- JWordNetManager: Implementação fornecida para serviço gramatical, usando WordNet;

- BDIManager: Interface para o gestor de Crenças, Desejos e Intenções. Permite adicionar ou remover crenças, recuperar crenças, adicionar pedidos do utilizador. Serviço usado para gerir a memória lógica do agente do utilizador (UA);

- BDIManagerImpl: Implementação fornecida para a arquitectura BDI através do componente JLups (ver Capítulo 7). Todo o modelo BDI encontra-se codificado com DLP através de regras LUPS;

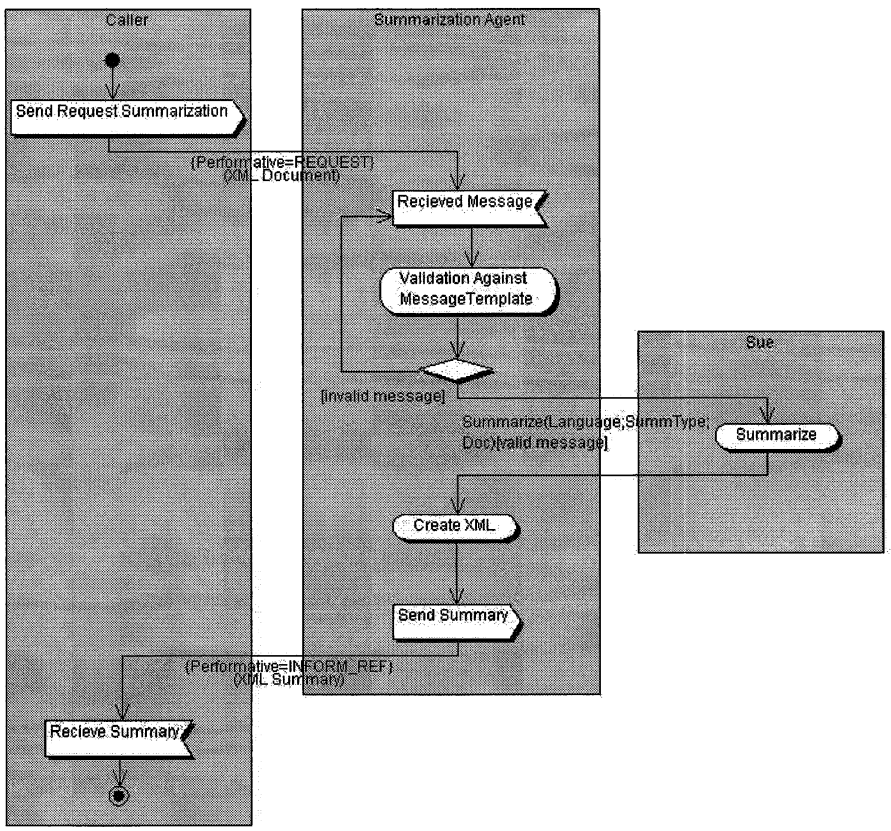

Figura 6.11: Agente para Sumarização de Documentos (DSA) - Diagrama de Actividade.

- SearchManager: Interface para o gestor de pesquisas, recebendo uma expressão e devolvendo uma lista de resultados encontrados. Este serviço é directamente usado por o agente de pesquisas (SA).
- GoogleSearchEngine: Implementação fornecida para o serviço de pesquisas, através do motor de pesquisa Google.
- Summarizer: Interface para a sumarização de documentos, que por sua vez é um serviço usado directamente pelo agente responsável pela sumarização de documentos (DSA);
- SummarizerImpl: Implementação fornecida para o serviço de sumarização, através do sumarizador extractivo Sue [Pal03].
- ManagerFactory: Fábrica dos serviços acima descritos, encapsulado a sua criação por parte dos seus clientes de forma a permitir um nível de escalabilidade maior (ver padrões de desenho nas secções 6.4.2.5 e 6.4.2.4).

Como classes auxiliares podem encontrar-se (ver figura 6.12):

- SearchItem: Item que representa uma pesquisa;
- Belief: Representação de uma crença;
- BusinessConfig: Classe que contém todas as configurações de negócio, usualmente parâmetros passados na inicialização do sistema. Resume-se a uma útil tabela de dispersão recuperada através do padrão de desenho *singleton*;

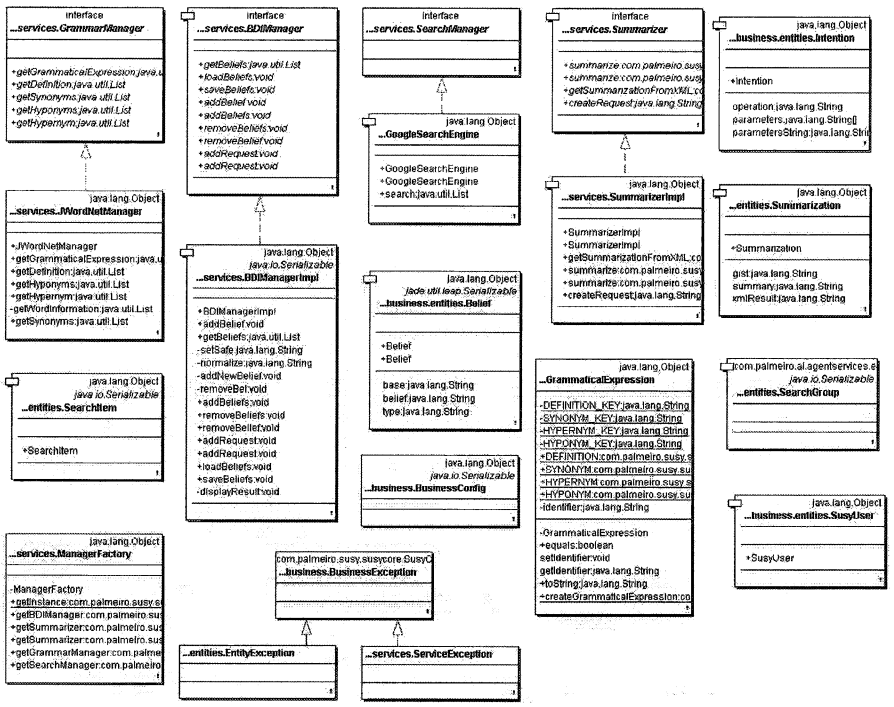

Figura 6.12: SusyCore - Diagrama de Classes.

- GrammaticalExpression: Representação de uma expressão gramatical;
- Intention: Representação de uma intenção assumida por um agente;
- Summarization: Resultado de uma sumarização;
- SearchGroup: Armazena um resultado parcial de uma pesquisa;
- SusyUser: Representação de um utilizador do sistema.

O mecanismo de excepções usado neste módulo é composto ainda por três classes (ver figura 6.12):

- BusinessException: Excepção base da camada de negócio;
- EntityException: Excepção ocorrida em *java beans* na camada das *entities*;
- ServiceException: Excepção gerada pela camada de serviços.

Ao nível dos agentes (*package com.palmeiro.susy.susycore.agents*) encontra-se um novo conjunto de classes dividido em três *sub-packages*. Na raíz desta estrutura encontram-se oito classes em que as mais importantes são abstracções para a classe *Agent*[38] (ver figura 6.13). Note-se que as classes que correspondem a agentes carregam os seus comportamentos (*behaviours*) na fase de inicialização.

- AbstractBaseAgent: Classe base dos agentes na plataforma. Esta classe extende directamente a classe *Agent*, fornecendo funcionalidades generalistas para todos as subclasses existentes na plataforma. Entre estas funcionalidades pode-se destacar o tratamento de erros genéricos assim como o mecanismo de *logging*;

[38] jade.core.Agent

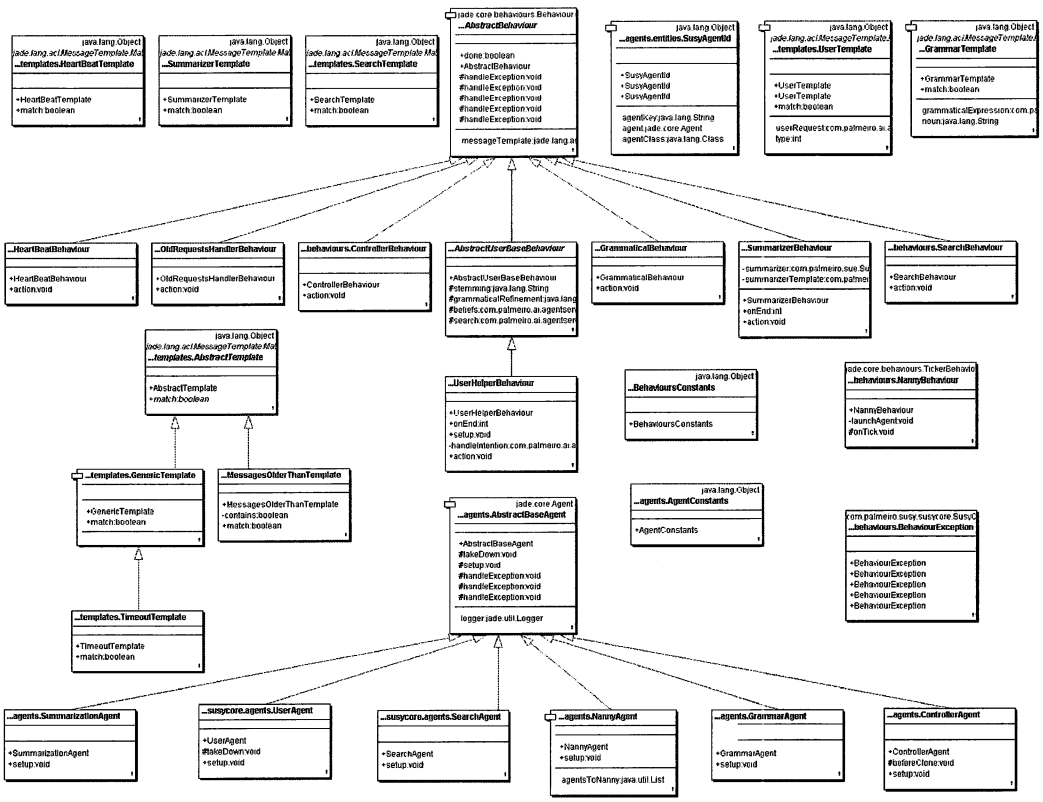

Figura 6.13: SusyAgents - Diagrama de Classes.

- AgentConstants: Contém diversas constantes usadas pelos agentes;
- ControllerAgent: Classe correspondente ao Agente Controller (CA);
- GrammarAgent: Classe correspondente ao Agente Gramatical (GA);
- NannyAgent: Classe correspondente ao Agente de Gestão de Serviços (NA);
- SearchAgent: Classe correspondente ao Agente de Pesquisa (SA);
- SummarizationAgent: Classe correspondente ao Agente de Sumarização (DSA);
- UserAgent: Classe correspondente ao Agente do Utilizador (UA).

No *package* de *behaviours* existem onze classes responsáveis pelos comportamentos dos agentes, encapsulando toda a lógica de negócio da aplicação (ver figura 6.13):

- AbstractBehaviour: Classe base dos comportamentos dos agentes na plataforma. Esta classe extende directamente a classe *Behaviour*[39], fornecendo funcionalidades generalistas para todos as subclasses. Entre estas funcionalidades podem-se destacar o tratamento de erros genéricos e mecanismo de *logging*;
- AbstractUserBaseBehaviour: Classe abstracta que fornece funcionalidades básicas para o comportamento do agente (UserHelperBehaviour), entre elas a comunicação com o agente de pesquisas (SA) e agente gramatical (GA), e acesso a crenças;
- BehaviourException: Excepção lançada pelos diversos "comportamentos";

[39] jade.core.behaviours.Behaviour

- ControllerBehaviour: Comportamento do agente *controller* (CA);
- GrammaticalBehaviour: Comportamento do agente gramatical (GA) usando o serviço *JWordNetManager* acima descrito;
- HeartBeatBehaviour: Comportamento usado pelos os agentes da Camada de Serviços de Negócio (ver secção 6.4.1.2) que fornecem serviços genéricos ao sistema (SA, GA e DSA). Este comportamento recebe um "sinal" do NA enviando uma resposta que representativa da sua sobrevivência;
- NannyBehaviour: Extende directamente a classe *TickerBehaviour*[40], enviando periodicamente um sinal para uma lista de agentes registados no sistema. Em caso de ausência de resposta este agente pode tentar relançar os agentes em falta;
- OldRequestsHandlerBehaviour: Comportamento usado pelo agente do utilizador dedicado ao tratamento de respostas antigas por parte dos serviços. Com este comportamento o agente pode tratar respostas que não chegaram no tempo estipulado (parâmetro configurável, com valor por defeito três segundos);
- SearchBehaviour: Comportamento responsável por pesquisas usando o serviço *SearchManager* descrito acima;
- SummarizerBehaviour: Comportamento responsável por sumarizações usando o componente de sumarização extractivo Sue;
- UserHelperBehaviour: Comportamento mais significativo da aplicação. Nele reside a lógica de interacção com a unidade de memória lógica (programa em JLups) através do serviço *BDIManager*. Este comportamento comunica com todos os agentes no sistema, incluindo o GA, SA, NA, DSA e o bastante próximo PA.

No *package* de *templates* existem nove classes responsáveis pela filtragem de mensagens recebidas pelos agentes (ver figura 6.13 e secção 6.4.4.4 sobre mecanismos de validação de dados):

- AbstractTemplate: *Template* base de validações de mensagens;
- GenericTemplate: *Template* genérica para validações de mensagens, podendo realizar um "*match*" com uma determinada *performative* e com um determinado agente através do seu *Agent Id*;
- GrammarTemplate: *Template* usada pelo GA de forma a filtrar mensagens num formato específico, com *performative* "QUERY-REF" e uma determinada estrutura de mensagem;
- HeartBeatTemplate: *Template* usada pelo comportamento *HeartBeatBehaviour* para receber *heartbeats* dos agentes de serviços;
- MessagesOlderThanTemplate: *Template* usada pelo comportamento *OldRequestsHandlerBehaviour* para receber somente mensagens antigas ou atrasadas, realizando um "*match*" temporal com o *timestamp* da mensagem;
- SearchTemplate: *Template* usada pelo *SearchBehaviour*;
- SummarizerTemplate: *Template* usada pelo *SummarizerBehaviour*;

[40] jade.core.behaviours.TickerBehaviour

- TimeoutTemplate: *Template* usada pelo *UserHelperBehaviour* de forma a receber mensagens num determinado período no tempo. Todas as mensagens mais velhas serão tratadas pelo *OldRequestsHandlerBehaviour*;

- UserTemplate: *Template* usada pelo *UserHelperBehaviour* para receber pedidos do PA.

No *package* de *entities* existe ainda uma entidade (*SusyAgentId*) que representa a identificação de um agente no sistema (ver figura 6.13).

6.4.6.3 Camada Genérica

Middleware de Agentes - Módulo AgentServices

Descrição: Módulo responsável pelo ciclo de vida da plataforma de agentes. Encapsula ainda componentes para gestão de agentes, assim como fornece uma *gateway* para comunicar com agentes disponíveis na plataforma.

Todas as implementações das interfaces principais associadas a este módulo são específicas da plataforma JADE.

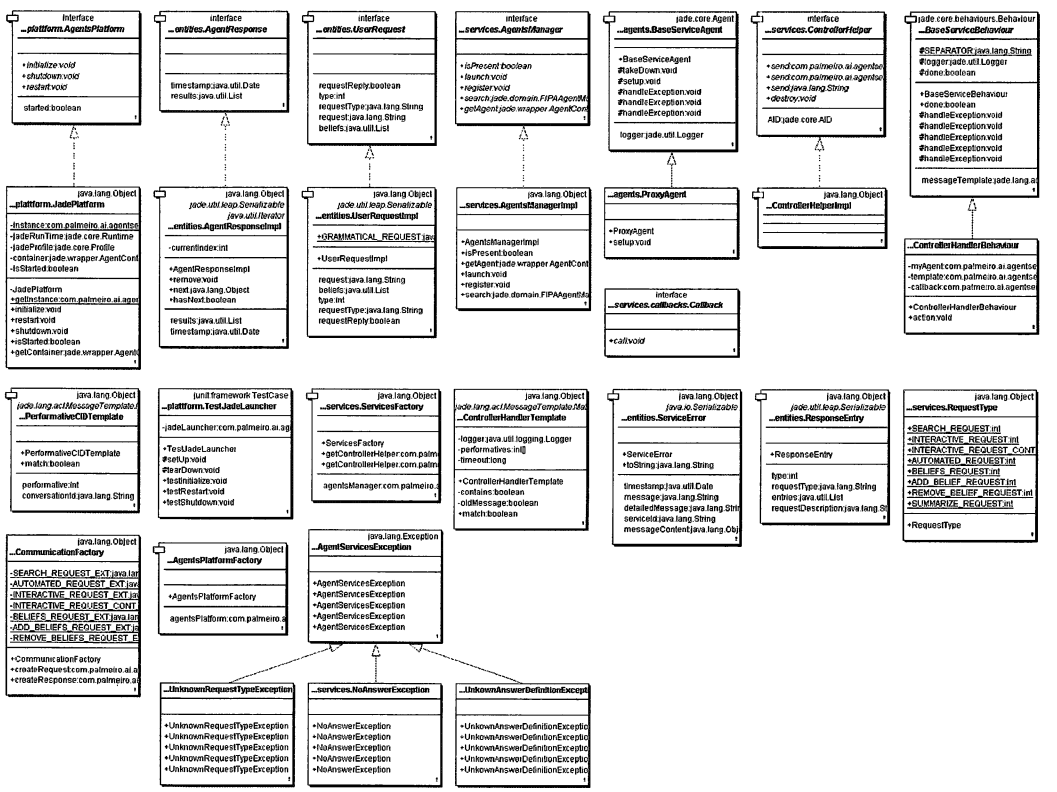

Figura 6.14: AgentServices - Diagrama de Classes.

Interface Pública: O presente módulo introduz um conjunto de classes (ver figura 6.14) para gestão e interacção com a plataforma de agentes, dentre as quais se destacam três interfaces principais.

AgentsPlatform: A interface define métodos para inicializar, reiniciar, parar e verificar o estado. A implementação fornecida é para a plataforma JADE.

```
public interface AgentsPlatform {
    public void initialize() throws AgentServicesException;
    public void shutdown() throws AgentServicesException;
    public void restart() throws AgentServicesException;
    public boolean isStarted();
}
```

AgentsManager: Um serviço para gestão de agentes que aglomera várias funcionalidades em forma de utilitário. Com este serviço é possível verificar se um agente está presente na plataforma, lançar e registar um agente no *Directory Facilitator*, pesquisar e recuperar um agente - respectivamente, pela ordem dos métodos abaixo descritos.

```
public interface AgentsManager {
    public boolean isPresent(String aid)
            throws AgentServicesException;
    public void launch(String aid, Agent agent)
            throws AgentServicesException;
    public void register(Agent agent, String serviceType, String serviceName)
            throws AgentServicesException;
    public DFAgentDescription search(Agent agent, String serviceType)
            throws AgentServicesException;
    public AgentController getAgent(String aid)
            throws AgentServicesException;
}
```

ControllerHelper: Este serviço encapsula a invocação directa de um agente (PA), tornando toda o camada de *Controller* independente da tecnologia de agentes.

```
public interface ControllerHelper {
    public AgentResponse send(UserRequest request, Agent destination)
            throws AgentServicesException;
    public AgentResponse send(UserRequest request, AID destinationAgent)
            throws AgentServicesException;
    public String send(String request, AID destinationAgent)
            throws AgentServicesException;
    public void destroy()
            throws AgentServicesException;
}
```

Classes Participantes: No presente módulo encontra-se um novo conjunto de classes dividido em quatro *subpackages*. Na raíz desta estrutura encontram-se apenas a classe representativa das excepções ocorridas e geradas no módulo.

No *package com.palmeiro.ai.agentservices.agents* existem duas classes juntamente com dois *subpackges*: *templates* e *behaviours* (ver figura 6.14):

- BaseServiceAgent: Classe base para os agentes do presente módulo;
- ProxyAgent: Agente Proxy (PA) funcionando como canal de comunicação com outros agentes;

No *package com.palmeiro.ai.agentservices.agents.behaviours* existem duas classes:

- BaseServiceBehaviour: Classe base de comportamentos de agentes do presente módulo;
- ControllerHandlerBehaviour: Comportamento do agente *proxy* (PA) para tratamento de mensagens atrasadas, considerando um conjunto alargado de *performatives* (por exemplo REJECT PROPOSAL, FAILURE, INFORM REF, INFORM,

entre outras). Este comportamento está intimamente ligado à aplicação *web* interagindo com ela através de uma interface de *callback* (descrita nos pontos que se seguem). A implementação fornecida para esta interface interage com a sessão *web* do utilizador, armazenando as mensagens recebidas para posterior consulta. Esta interface permite o uso do padrão *callback* para interacção com qualquer tipo de tecnologia;

No *package com.palmeiro.ai.agentservices.agents.templates* existem duas classes:

- ControllerHandlerTemplate: Filtro do PA para recepção de mensagens baseado nas *performatives* descritas no comportamento *ControllerHandlerBehaviour*;
- PerformativeCIDTemplate: Filtro de mensagens baseado numa *performative* e numa identificação de conversa, usado pelo PA para receber somente respostas a conversações iniciadas por o próprio;

No *package com.palmeiro.ai.agentservices.entities* existem seis classes:

- AgentResponse: Interface que encapsula uma resposta do componente *ControllerHelper*, devolvendo uma lista de resultados (objectos *ResponseEntry*) e um *timestamp* da resposta;
- AgentResponseImpl: Implementação fornecida para a interface acima descrita;
- ResponseEntry: Resultado concreto de um agente, como resposta a uma mensagem. Contém o tipo de pedido, o tipo de resposta, uma descrição da resposta e uma lista de possíveis registos encapsulados nesta resposta;
- ServiceError: Classe que contém uma mensagem atrasada (na sua forma original) ou uma descrição de uma possível falha. Usada pelo comportamento *ControllerHandlerBehaviour* que está detalhado na secção da "Realização Interna das Operações";
- UserRequest: Parâmetro de *Input* nas operações de envio de mensagens para agentes, no componente *ControllerHelper* que também está detalhado na secção dedicada à "Realização Interna das Operações". Esta classe tem como atributos principais um tipo de pedido (bem conhecido no sistema de forma a ser bem interpretado), se o pedido exige resposta (*request-reply*), e o pedido em si. Este objectos são construídos por classes do tipo *ComunicationFactory*;
- UserRequestImpl: Implementação da classe acima descrita.

No *package com.palmeiro.ai.agentservices.platform* existem quatro classes:

- AgentsPlatform: Interface introduzida na secção "Interface Pública", sendo um componente mais significativo do presente módulo. Nela podemos encontrar métodos para inicializar, parar e reinicializar o sistema, assim como verificação do estado do sistema. É seu objectivo principal ocultar uma determinada plataforma de agentes, fornecendo métodos de gestão bastante genéricos;
- AgentsPlatformFactory: *Factory* responsável pela criação do objecto que gere a plataforma de agentes;
- JadePlatform: Implementação fornecida para a interface *AgentsPlatform* incidindo sobre a plataforma de agentes JADE[41]. É usada o padrão *singleton* para a criação de um objecto deste tipo;

[41] http://jade.tilab.com/

- TestJadeLauncher: Teste JUnit à interface *AgentsPlattform*, neste caso à implementação *JadePlattform*.

No *package com.palmeiro.ai.agentservices.services* existem dez classes:

- AgentsManager: Serviço para gestão de agentes, contento operações para registo, lançamento, verificação de estado, pesquisa e recuperação de agentes. As funcionalidades mais interessantes encontram-se detalhadas na secção seguinte (Realização Interna das Operações);

- AgentsManagerImpl: Implementação para a plataforma JADE, usando o componente *AgentsPlatform* para aceder à plataforma de agentes. Ao nível da implementação esta classe apresenta-se apenas como um *wrapper* para classes da plataforma JADE que encapsula de forma a fornecer funcionalidades mais directas e intuitivas;

- ComunicationFactory: Classe que implementa o padrão *factory* sendo responsável pela criação de objectos *UserRequest* que por sua vez representam uma mensagem a enviar pelo componente *ControllerHelper*. Nesta classe existem vários tipos de pedidos bem conhecidos pela plataforma;

- ControllerHelper: Interface para envio de mensagens para agentes. Encontra-se descrita na secção Realização Interna das Operações. Na sua essência funciona como um *proxy* de comunicação com uma plataforma de agentes;

- ControllerHelperImpl: Implementação da interface acima descrita;

- NoAnswerException: Excepção gerada pelo *ControllerHelper* quando há ausência de resposta de um agente a um pedido (*UserRequest*) que apresentava uma qualidade de serviço *request-reply*;

- RequestType: Classe que mantém os tipos de pedidos conhecidos pelo sistema, sendo usada pela *ComunicationFactory*;

- ServicesFactory: Classe que implementa o padrão *factory* sendo responsável pela criação dos principais serviços como *ControllerHelper* e *AgentsManager*;

- UnknownRequestTypeException: Excepção gerada pela *ComunicationFactory* quando o tipo de pedido não é conhecido pelo sistema;

- UnkownAnswerDefinitionException: Excepção gerada pelo componente *ControllerHelper* quando uma resposta não apresenta um tipo de resposta conhecido ou não pôde ser interpretada.

No *package com.palmeiro.ai.agentservices.services.callbacks* existe uma classe:

- Callback: Interface que permite a invocação - via *callback* - de sistemas externos quando uma mensagem atrasada ou em falha surge no sistema. É directamente usada pelo comportamento *ControllerHandlerBehaviour*.

No *package com.palmeiro.ai.agentservices* existe uma classe:

- AgentServicesException: Excepção base do módulo.

Realização Interna das Operações: Esta secção descreve as operações internas a este módulo, consideradas mais relevantes para a arquitectura. Nomeadamente:

- *ControllerHelper*: Inicialização do componente;
- *ControllerHelper*: Interacção com outros agentes;
- *ControllerHelper*: Tratamento de mensagens antigas, excepções e falhas;
- *AgentsManager*: Pesquisar por um agente;
- *AgentsManager*: Lançar um agente;
- *AgentsManager*: Registar um agente.

***ControllerHelper* - Inicialização:** O diagrama de sequência da figura 6.15 exemplifica a forma de inicialização deste componente. Note-se o uso do componente *AgentsManager* para a verificação da presença o agente PA, assim como para o seu lançamento. Caso o agente já esteja presente no sistema não é relançado nesta inicialização. Aquando do lançamento do agente PA é realizado o registo (no PA) do comportamento para tratamento de mensagens, assim como os filtros associados para recepção de mensagens.

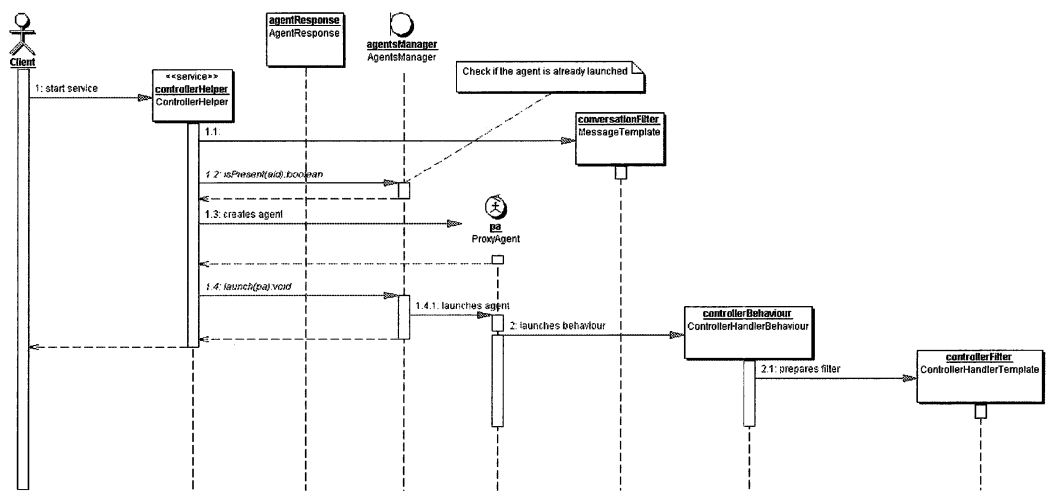

Figura 6.15: ControllerHelper: Inicialização - Diagrama de Classes.

***ControllerHelper* - Interacção com outros Agentes:** O diagrama de sequência da figura 6.16 exemplifica a forma como este componente interage com outros agentes. Note-se que os parâmetros principais da operação responsável por esta funcionalidade são o agentes destino e a mensagem em causa. Com esta mensagem (contida num objecto *UserRequest*), é criada uma mensagem ACL temporária contendo o pedido do cliente e também o seu agente destino. É realizado o envio da mensagem e espera-se alguns segundos[42] por uma resposta, devolvida ao cliente na forma de um objecto *UserReponse*. Caso exista um *timeout* na recepção da mensagem, esta poderá eventualmente ser tratada pelo comportamento *ControllerHandlerBehaviour* descrito na secção seguinte.

[42]Valor configurável.

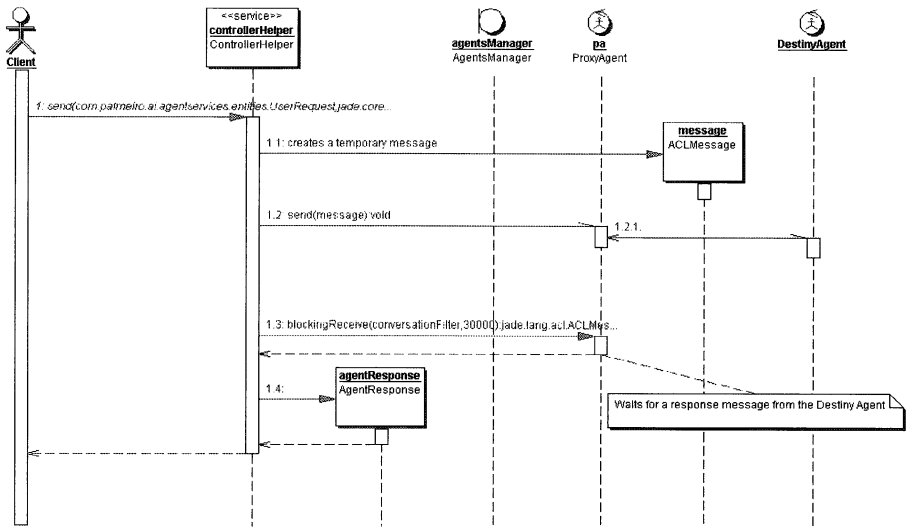

Figura 6.16: ControllerHelper: Envio de uma mensagem - Diagrama de Sequência.

***ControllerHelper* - Tratamento de Mensagens:** O diagrama de sequência da figura 6.17 exemplifica a forma como este componente realiza o tratamento de mensagens antigas ou falhas. Esta funcionalidade usa uma interface de *callback* para interagir com outros componentes, que no presente contexto são as sessões *web*[43] dos utilizadores. Quando é recebida uma mensagem que passe pelo filtro *ControllerHandlerTemplate* e tenha uma *performative* FAILURE ou EXCEPTION a mensagem será directamente inserida na sessão através de um *callback*. Caso a mensagem apresente a *performative* INFORM REF será convertida para um objecto *ServiceError* que será igualmente inserido na sessão através do mesmo processo. Note-se que a aplicação *web* disponibiliza uma página para consulta deste tipo de mensagens, podendo o utilizador consultar mensagens antigas e analisar os erros ou falhas ocorridas no sistema.

***AgentsManager* - Lançar um agente:** O diagrama de sequência da figura 6.18 exemplifica a forma como o componente *AgentsManager* lança um agente na plataforma JADE. O fluxo apresenta-se bastante simples e recorre ao uso do objecto *AgentContainer* disponibilizado pela plataforma JADE para "aceitar" um novo agente no sistema.

***AgentsManager* - Registar um agente:** O diagrama de sequência da figura 6.19 exemplifica o processo de registar um agente na plataforma. O fluxo é iniciado através da criação de um objecto *DFAgentDescription*[44] para manter a descrição do agente em causa, neste caso o identificador único do agente (Agent ID, ou AID). Cria-se posteriormente uma descrição de serviço (através da classe *ServiceDescrition*[45]) contendo o nome e tipo de serviço (parâmetros de entrada), e o protocolo associado (FIPA, neste caso). Associa-se esta descrição ao agente através da classe *DFService*[46] que funciona como um *proxy* para o agente *Directory Facilitator*.

***AgentsManager* - Pesquisar por um agente:** O diagrama de sequência da figura 6.20 exemplifica o processo de pesquisa por um agente. O fluxo é iniciado através

[43] Através do objecto javax.servlet.http.HttpSession .
[44] Plataforma JADE, *package jade.domain.FIPAAgentManagement.DFAgentDescription*
[45] Plataforma JADE, *package jade.domain.FIPAAgentManagement.ServiceDescrition*
[46] Plataforma JADE, *package jade.domain.DFService*

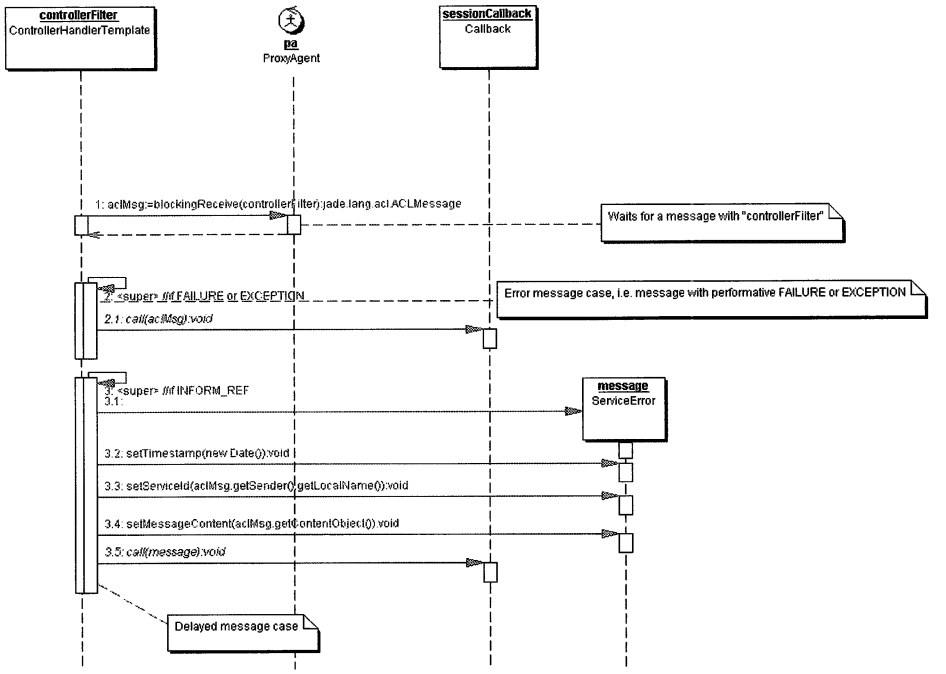

Figura 6.17: ControllerHelper: Tratamento de mensagens atrasadas ou falhas - Diagrama de Sequência.

da criação de uma descrição do agente (*DFAgentDescription*) e de uma descrição de serviço para pesquisa (*ServiceDescrition*), de forma semelhante ao fluxo anterior. Como atributos de pesquisa (*SearchConstraints*[47]) é apenas usado o número de resultados. Por fim é realizada a pesquisa através do objecto *DFService* que devolve uma lista de resultado (objectos *DFAgentDescription*). Se for encontrado um resultado com o mesmo tipo de serviço que o usado no parâmetro de entrada, então o agente em causa encontra-se registado no sistema. Caso esta lista esteja vazia ou não exista nenhum agente com a descrição pedida, é porque o agente não se encontra registado.

Segurança - Módulo SusySecurity

Descrição: O presente módulo é dedicado à segurança da plataforma *web*. Contém serviços auxiliares para tratamento do mecanismo de autorização, assim como uma implementação de um *realm* de Tomcat[48] para tratamento de perfis (*roles*) dos diferentes utilizadores.

Classes Participantes: Existem cinco classes disponíveis neste módulo:

- SusySecurityManager: Interface do componente responsável pela gestão de segurança da aplicação *web*, como por exemplo gestão dos perfis dos utilizadores;
- SusySecurityManagerImpl: Implementação fornecida para o componente de segurança da plataforma;
- SecurityConstants: Classe que armazena valores constantes reutilizáveis;

[47] Plataforma JADE, *package jade.domain.FIPAAgentManagement.SearchConstraints*
[48] Ver http://tomcat.apache.org/

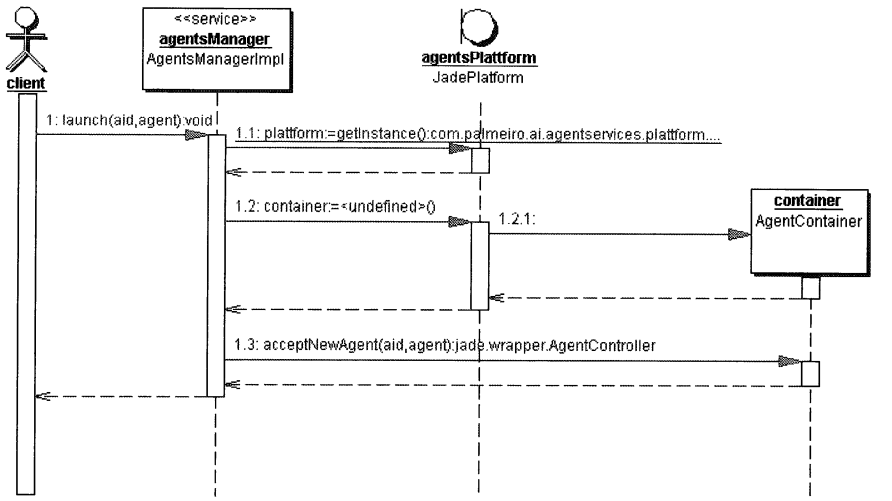

Figura 6.18: AgentsManager: Lançar um agente - Diagrama de Sequência.

- SecurityFactory: Fábrica de serviços do presente módulo, encapsulando a sua criação por parte dos seus clientes de forma a permitir um nível de escalabilidade maior (ver padrões de desenho 6.4.2.5 e 6.4.2.4).

- TomcatRealm: Implementação fornecida de um *realm* Tomcat específico para a plataforma usando o serviço acima definido.

Realização Interna das Operações: Não aplicável.

6.4.7 Realização das Funcionalidades Mais Significativas

Esta secção aborda o desenho detalhado das funcionalidades consideradas como mais significativas para a arquitectura de software, e identificadas na secção "6.3.4 Casos mais Significativos para a Arquitectura", na Vista Funcional.

Note-se que existem vários outros fluxos arquitecturais significativos na aplicação, que embora não estejam detalhados nesta secção já foram previamente abordados como "Elementos Mais Significativos do Desenho"(secção 6.4.6, subsecção 6.4.6.3 Camada Genérica).

Estas funcionalidades estão fisicamente representadas no núcleo da aplicação, módulo SusyCore.

6.4.7.1 Sumarização de Documentos

Descrição

A presente secção descreve de forma detalhada o fluxo de sumarização de um documento. Este fluxo é considerado significativo para a arquitectura pois descreve como é realizada a sumarização de documentos, quais os componentes usados e as opções disponíveis. Todos os serviços (agentes) restantes seguem a mesma linha de desenho.

Estrutura

A estrutura do presente fluxo é constituída por sete classes e duas interfaces:

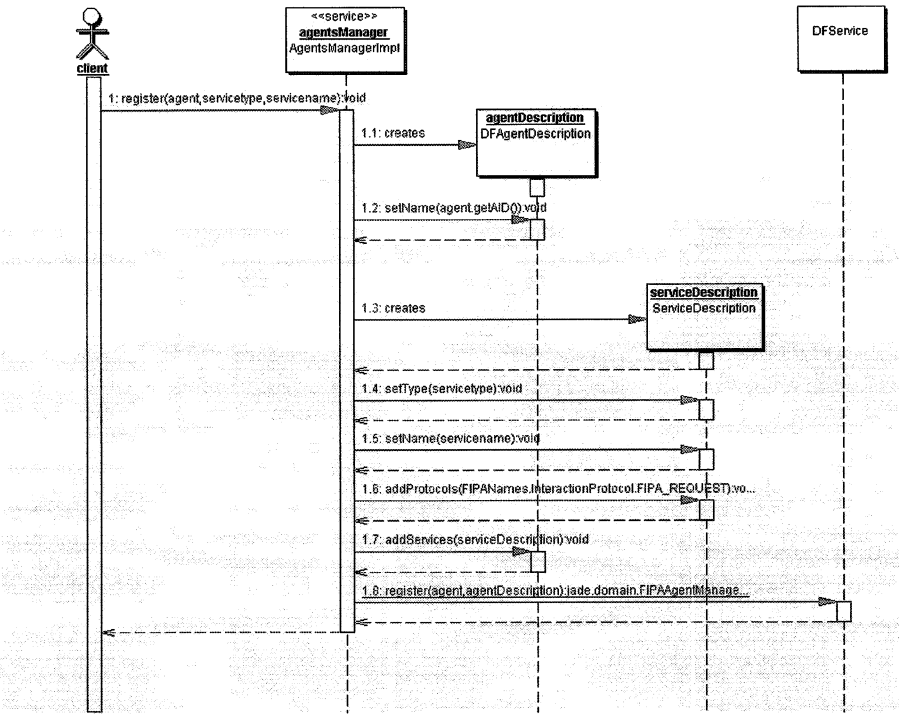

Figura 6.19: AgentsManager: Registar um agente - Diagrama de Sequência.

- *ControllerHelper*;
- *ControllerHelperImpl*;
- *ManagerFactory*;
- *Summarizer*;
- *SummarizerImpl*;
- *AbstractBaseAction*;
- *SummarizationAction*;
- *SummarizationForm*;
- *Summarization.*

Cada uma destas classes e interfaces está descrita na secção 6.4.6, que deverá ser consulta para mais detalhes.

O diagrama de classes está descrito na figura 6.22.

Cenários

Uma sumarização de um documento engloba dois cenários principais do ponto de vista arquitectural: Pedido do utilizador via *web* e sumarização do documento por parte do agente especializado em sumarizações (DSA).

Um utilizador autenticado no sistema acede à funcionalidade de sumarização de documentos. Nesta página escolhe o tipo de sumarização desejado (sumarização por *Term-Frequency Inverse-Sentence-Frequency* (TF-ISF) ou por *keywords*), e a língua sobre a

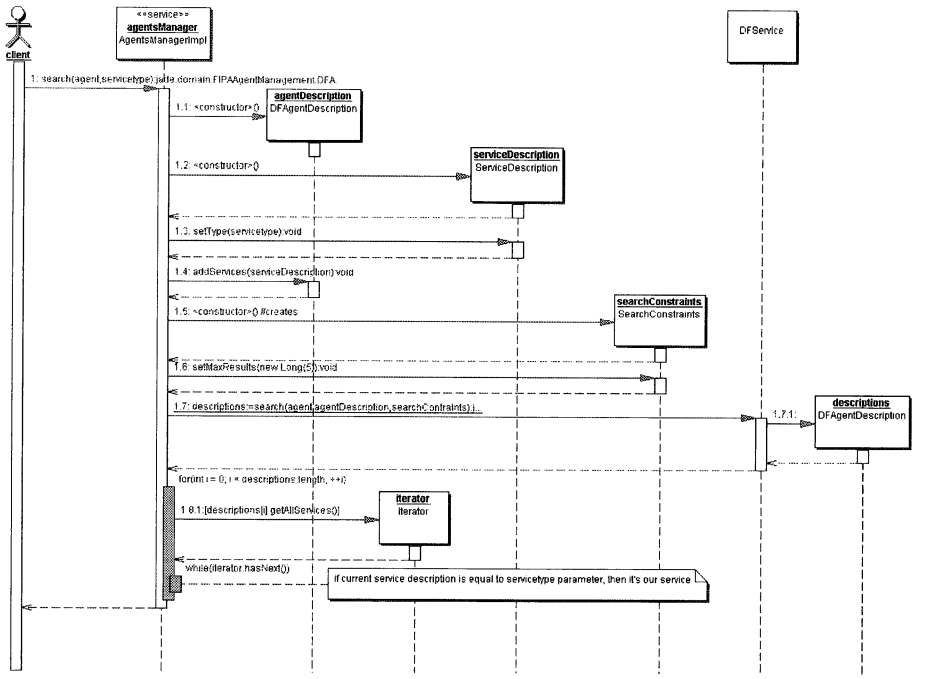

Figura 6.20: AgentsManager: Pesquisar por um agente - Diagrama de Sequência.

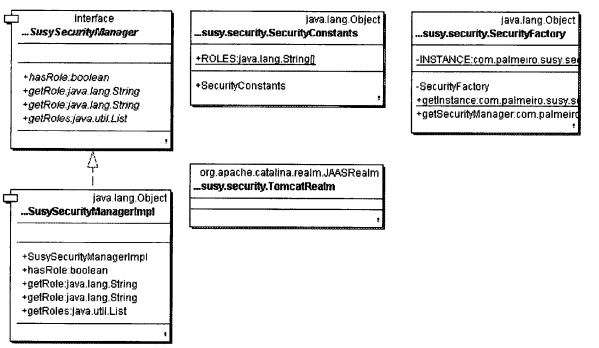

Figura 6.21: SusySecurity - Diagrama de Classes.

qual deseja que a sumarização seja feita. Insere o documento (texto) na área reservada e invoca a operação através do *link summarize*.

A acção Struts (*SummarizationAction*) responsável pelo controlo desta funcionalidade recebe o pedido do utilizador e retira do *ActionForm* os valores (tipo de sumarização, língua e documento) introduzidos pelo utilizador. Através de um serviço especializado e de apoio à sumarização cria uma mensagem XML representante do pedido a realizar ao agente. Note-se que a construção deste serviço é delegada a um *business delegate* (*ManagerFactory*) que fica responsável pela construção de todo o tipo de serviços presente neste módulo, escondendo as classes que implementam as interfaces dos respectivos serviços. Após a construção da mensagem a acção recupera o objecto (*ControllerHelper*, ver secção "6.4.6.3 *Middleware* de Agentes - Módulo AgentServices", e figura 6.16) responsável pelo canal de comunicação para os agente. Este objecto representa um canal que contém um agente *proxy* (PA), único para cada utilizador e respectiva sessão, que se responsabiliza pela comunicação com os restantes agentes do sistema. Através do objecto *ControllerHelper* a acção envia a mensagem XML para o

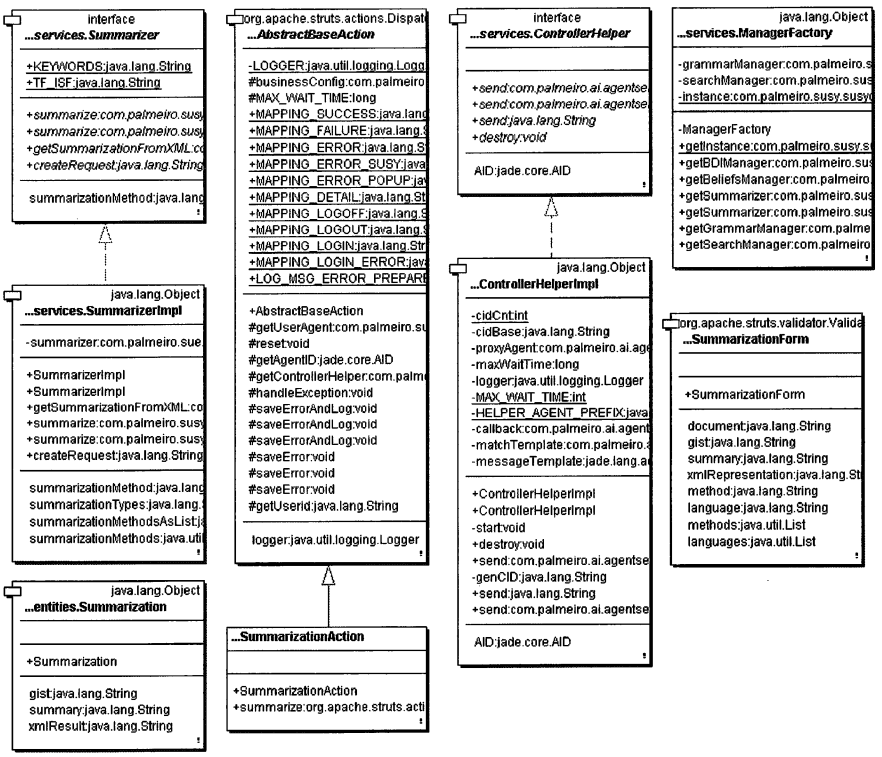

Figura 6.22: Processo de Sumarização - Diagrama de Classes.

DSA. Este fluxo encontra-se representado na figura 6.23.

Após o envio de uma mensagem XML para o DSA, este valida-a com base na *template* (filtros de mensagens JADE) destinada para filtragem de mensagens. Se a mensagem for válida, o agente realiza um *parse* ao XML extraindo o documento, tipo de sumarização pretendida e língua para sumarização. Após ter os conteúdos base de uma sumarização, o agente invoca o serviço de sumarização Java presente no sistema (Sue). Quando este serviço retornar um resultado, será criada uma mensagem XML contendo uma frase identificadora do documento, assim como o sumário pedido. Caso sejam detectados erros este agente irá enviar uma mensagem de erro para o PA, que se encarrega de tratar a excepção ou falha (ver secção 6.4.6.3 e figura 6.17). Este fluxo encontra-se representado na figura 6.24.

A acção ao receber o resultado (retornado pelo *ControllerHelper* com origem no DSA) da sumarização, usa o serviço de apoio à sumarização para recuperar os valores mensagem XML (sumário e *gist*), inserindo-os no escopo de *request* e realizando um *forward* para a página de sumarização onde serão apresentados os resultados.

6.4.7.2 Agente do Utilizador, Ciclo de Vida

Descrição

Este fluxo é considerado significativo para a arquitectura pois descreve todo o processo de autenticação de um utilizador no sistema, podendo ser analisado em detalhe a inicialização dos agentes associados assim como a finalização e limpeza do sistema aquando de um *logout*.

Serão abordados os seguintes cenários: Autenticação e inicialização dos subsistemas associados ao utilizador, e *Logout* e finalização dos subsistemas associados.

Neste cenários existirão várias referências a componentes do módulo AgentServi-

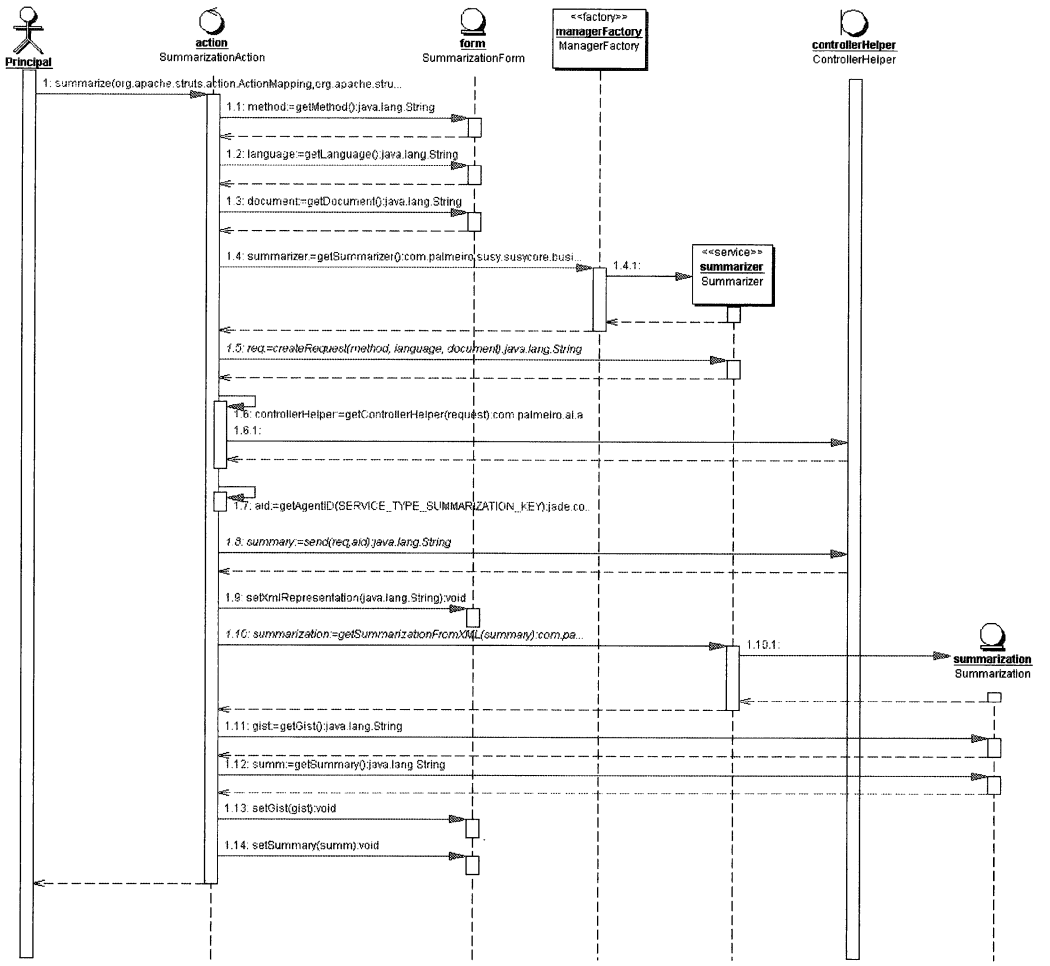

Figura 6.23: Sumarização de um Documento via *Web* (1) - Diagrama de Sequência.

ces, que por apresentarem o estereótipo *boundary* não serão descritos em detalhe nos cenários nem nos diagramas de sequência associados. Para mais informações sobre estes componentes deve ser consultada a secção 6.4.6.3).

Estrutura

A estrutura do presente fluxo é constituído por onze classes e quatro interfaces:

- *AbstractBaseAction*;
- *AuthenticationAction*;
- *ManagerFactory*;
- *BDIManager*;
- *BDIManagerImpl*;
- *UserAgent*;
- *UserHelperBehaviour*;
- *OldRequestsHandlerBehaviour*.

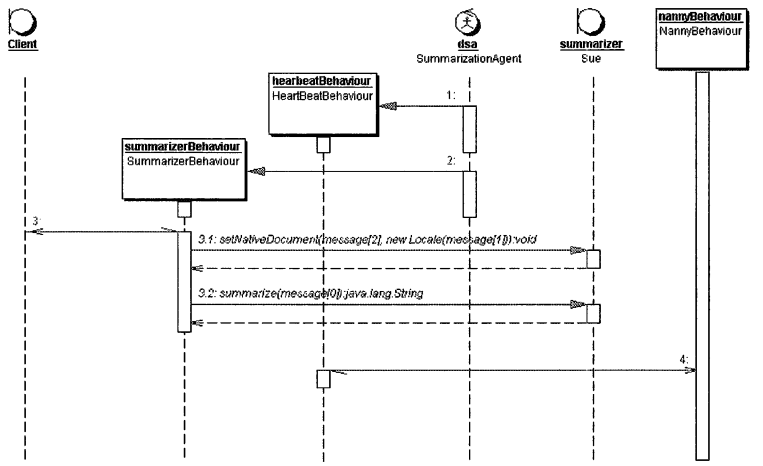

Figura 6.24: Sumarização de um Documento via Agente (2) - Diagrama de Sequência.

- *ControllerHelper*;
- *ControllerHelperImpl*;
- *AgentsManager*;
- *AgentsManagerImpl*;
- *ServicesFactory*;
- *JLups*;
- *JLupsImpl*.

Cada uma destas classes e interfaces está descrita na secção 6.4.6, que deverá ser consultada para mais detalhes.

O diagrama de classes está descrito na figura 6.25.

Cenários

Autenticação e Inicialização dos Subsistemas: Após uma correcta autenticação no sistema através do módulo JAAS instalado, o controlo do fluxo passa para a acção Struts de autenticação que se responsabiliza pela inicialização de todos os subsistemas associados à sessão do utilizador.

O primeiro subsistema a ser lançado é o componente *ControllerHelper*, que é um *proxy* para a plataforma de agentes presente no sistema. Este componente é inicializado com os seguintes parâmetros: nome do utilizador, que ficará associado ao UA; tempo máximo de espera por uma mensagem, que é um atributo configurável; e a passagem de uma implementação (*UserSessionCallback*) da interface *Callback*, que se responsabiliza por inserir os resultados de uma falha ou de mensagens antigas não recebidas na sessão *web* (ver secção 6.4.6.3), quando invocada pelo *ControllerHelper*.

Após a inicialização deste componente, será lançado o UA através do componente *AgentsManager*, cuja construção é delegada à fábrica de serviços (*ServicesFactory*) existente no módulo AgentServices. Com o apoio do componente *AgentsManager* a acção lança o UA, e insere na sessão *web* o nome associado ao UA para posteriores contactos e finalizações. O UA ao ser inicializado adiciona automaticamente dois comportamentos:

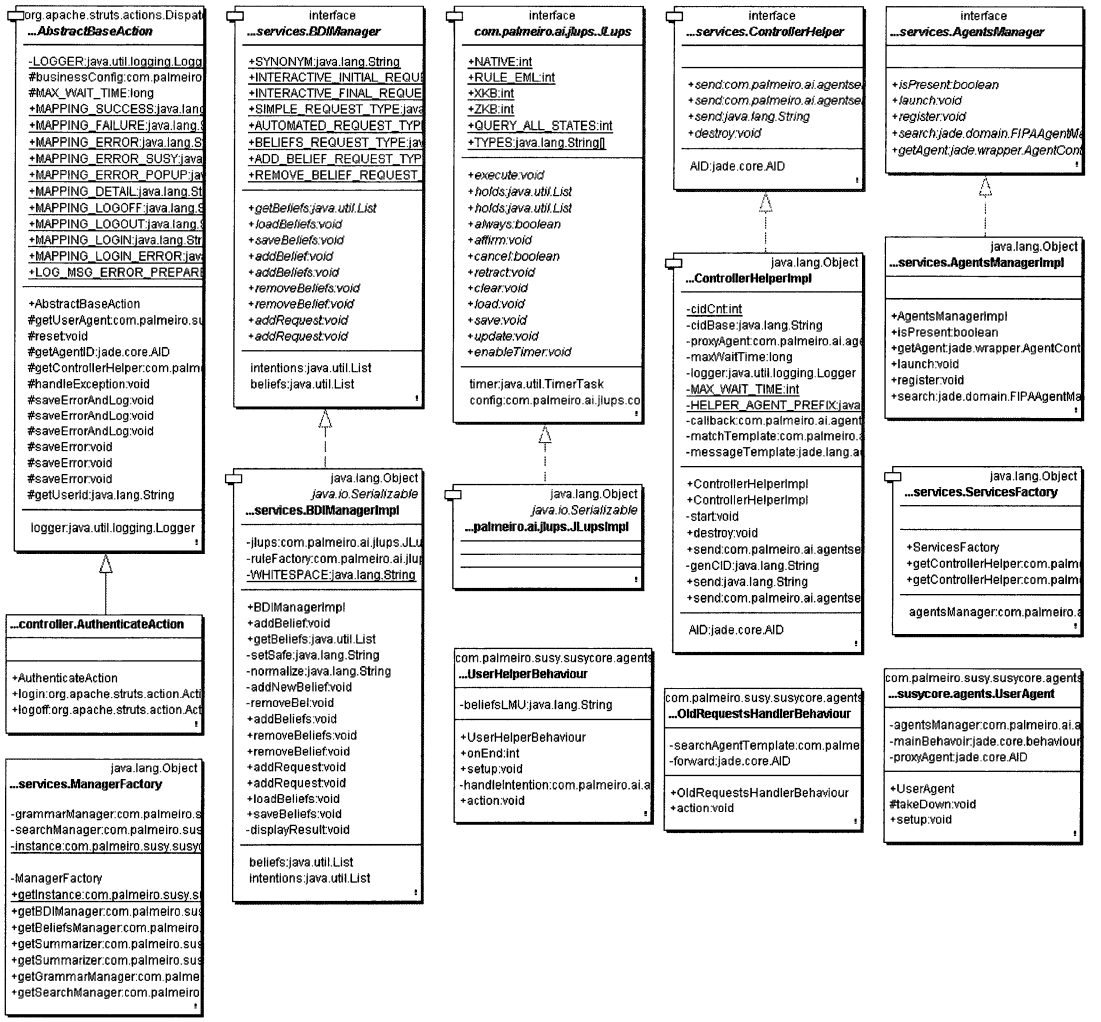

Figura 6.25: Agente do Utilizador e respectivo ciclo de vida - Diagrama de Classes.

- *UserHelperBehaviour*: irá conter todas a regras do comportamento do agente face a estímulos externos;
- *OldRequestsHandlerBehaviour*: irá conter regras para tratamento de mensagens antigas, erros ou falhas.

Na inicialização do comportamento *UserHelperBehaviour*, este vai construir o componente *BDIManager* através da fábrica de serviços (*ManagerFactory*) existente no módulo SusySore. Este componente será responsável pela gestão de crenças do UA, usando o componente JLups (descrito no Capítulo 7) para apoio ao tratamento e gestão de regras associadas ao agente. No inicio de vida de um agente, será fornecido um conjunto de regras de negócio que irão modelar todo o comportamento do respectivo UA. Em posteriores reinicializações o agente irá carregar sempre a mesma base de conhecimento, que irá ficar cada vez mais completa consoante o nível de interacção entre utilizador e UA. O conhecimento armazenado será sempre usado em posteriores interacções, influenciando o fluxo normal dos acontecimentos. No entanto todo o comportamento do JLups está escondido dentro de um componente que se responsabiliza de fornecer ao agente uma interface mais orientada para crenças, desejos, e intenções, sendo que a implementação usada segue linhas orientadas a uma arquitectura BDI, não implementado uma solução BDI concreta nem completa.

Este fluxo está representado do diagrama de sequência da figura 6.26.

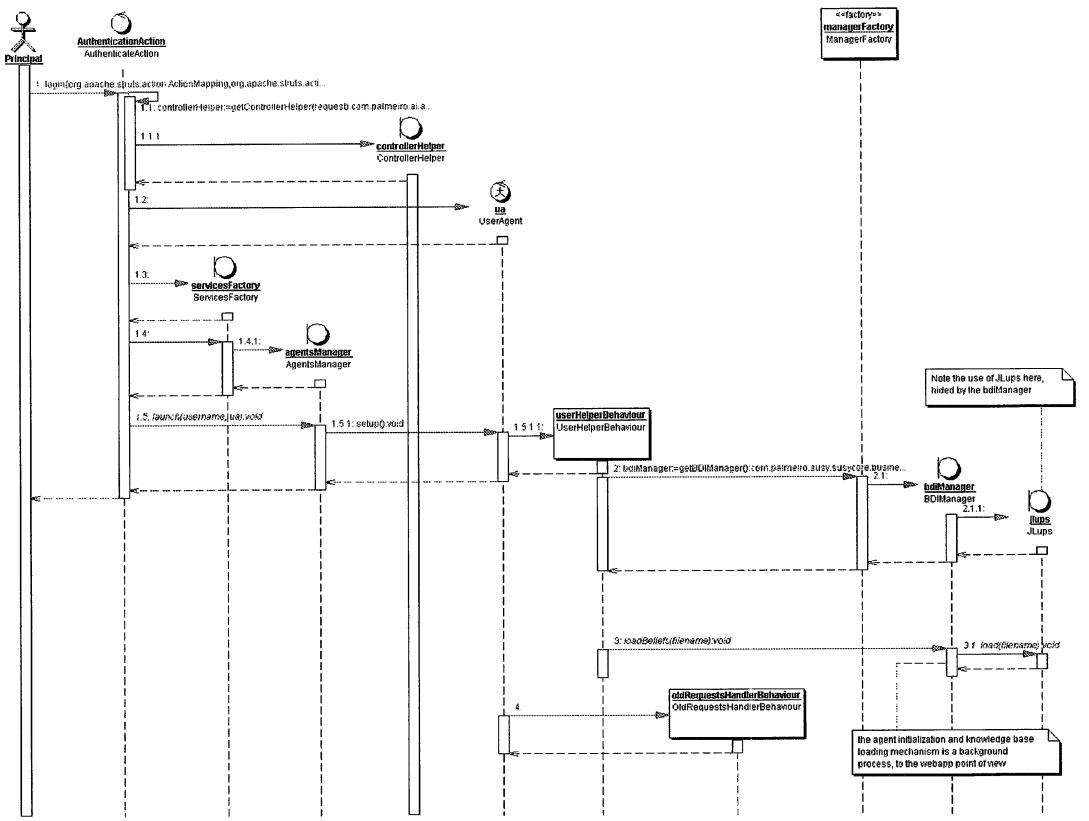

Figura 6.26: Login e Inicialização do UA - Diagrama de Sequência.

***Logout* e Finalização dos Subsistemas:** Após o utilizador pressionar o *link* para *logout*, o controlo do fluxo passa novamente para a acção Struts de autenticação, que analogamente ao cenário anterior, também se responsabiliza pela finalização de todos os subsistemas associados à sessão do utilizador.

Novamente através do componente *AgentsManager* a acção destrói o UA, fazendo com que um processo em cascata se inicie do lado do agente. Por sua vez quando o agente detecta que vai ser interrompido o seu ciclo de vida, envia sinais para todos os seus comportamentos se prepararem para a finalização. Nesta finalização o componente *UserHelperBehaviour* grava toda a informação retida na presente sessão para a base de conhecimento através do componente *BDIManager*, que por sua vez invoca o método de gravação do JLups.

De forma a pôr fim à sessão, a acção Struts invoca finalmente o método de destruição do componente *ControllerHelper*, fazendo com que o agente *proxy* associado à sessão (embora encapsulado no *ControllerHelper*) também se finalize. Como passo final a sessão *web* do utilizador é invalidada.

Este fluxo está representado do diagrama de sequência da figura 6.27.

6.4.7.3 Agente do Utilizador, Gestão de Crenças e Intenções

Descrição

Este fluxo é considerado significativo para a arquitectura pois descreve o núcleo de processamento do Agente Pessoal (UA), desde a recepção de pedidos do utilizador, até à

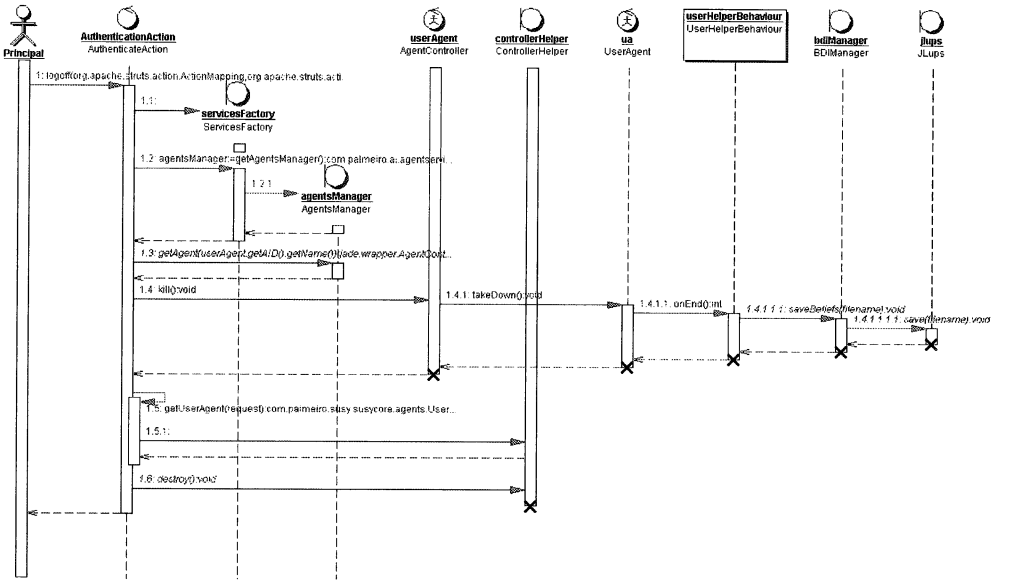

Figura 6.27: Logout e Finalização do UA - Diagrama de Sequência.

sua realização. Este processo inclui a gestão crenças e detecção de intenções e descreve a forma como o utilizador interage com a qualidade da informação e com o próprio sistema em si.

Estrutura

A estrutura do presente fluxo é constituído por treze classes e seis interfaces (ver figura 6.28):

- *AbstractBaseAction*;
- *AgentResponse*;
- *AgentResponseImpl*;
- *Beliefs*;
- *BeliefsAction*;
- *BeliefsForm*;
- *BDIManager*;
- *BDIManagerImpl*;
- *CommunicationFactory*;
- *ControllerHelper*;
- *ControllerHelperImpl*;
- *Intention*;
- *JLups*;
- *JLupsImpl*;

- *JLupsRuleFactory*;
- *UserAgent*;
- *UserHelperBehaviour*;
- *UserRequest*;
- *UserRequestImpl.*

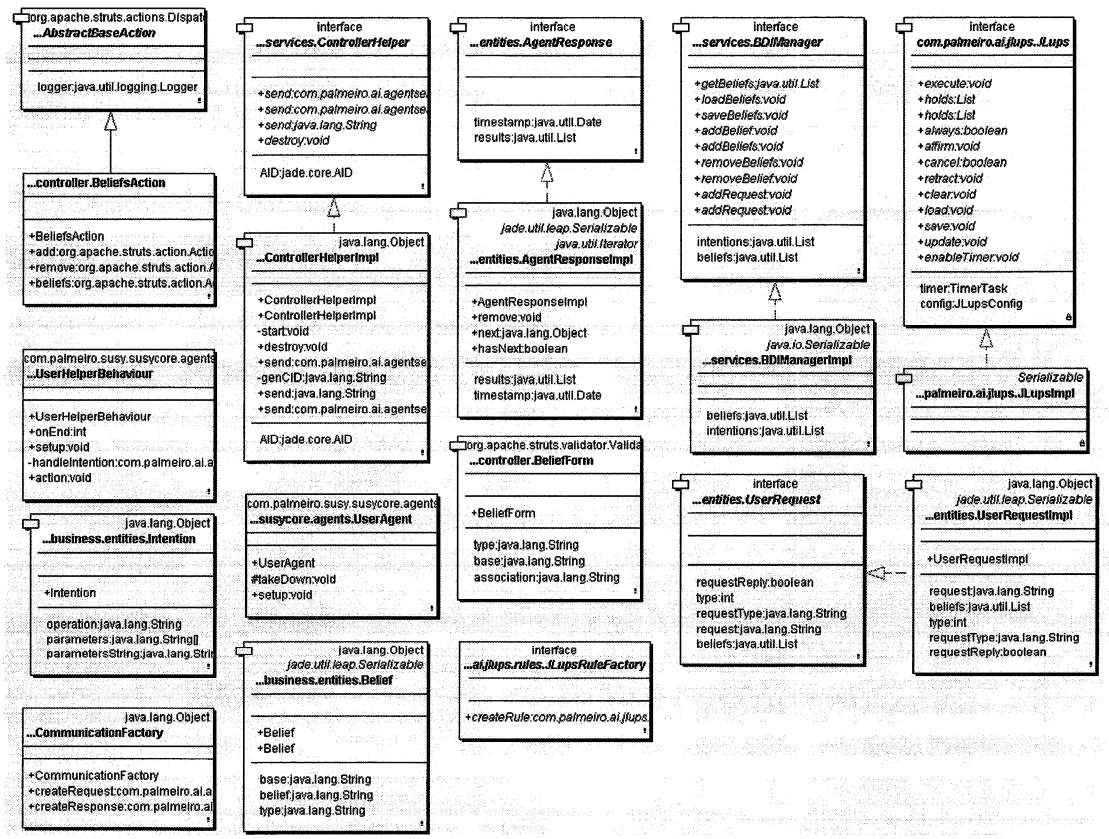

Figura 6.28: Listagem das Crenças do UA - Diagrama de Classes.

Cenários

Listagem de Crenças: O fluxo é iniciado através de um pedido de listagem de crenças por parte do utilizador. A acção (*BeliefsAction*) responsável pelas crenças (ver figura 6.29) assume o controlo do fluxo, e através de um serviço de apoio à comunicação (*CommunicationFactory*) é construída uma mensagem associada a um pedido de recuperação de crenças. Recupera-se o componente *ControllerHelper* mantido em sessão e envia-se esta mensagem para o UA.

O comportamento principal do UA (*UserHelperBehaviour*) ao receber a mensagem (ver figura 6.30) proveniente do *ControllerHelper* retira do seu conteúdo o pedido do utilizador, encapsulado-o num objecto *UserRequest*. O comportamento adiciona o pedido e o tipo de pedido à sua base de conhecimento, através do componente *BDIManager*. Este componente encarrega-se de criar uma regra JLups através da fábrica de regras *JLupsRuleFactory* disponibilizada pelo JLups, e adiciona-a através de um *assert* (método *affirm*), seguido do comando de *update* para actualizar toda a base de conhecimento de

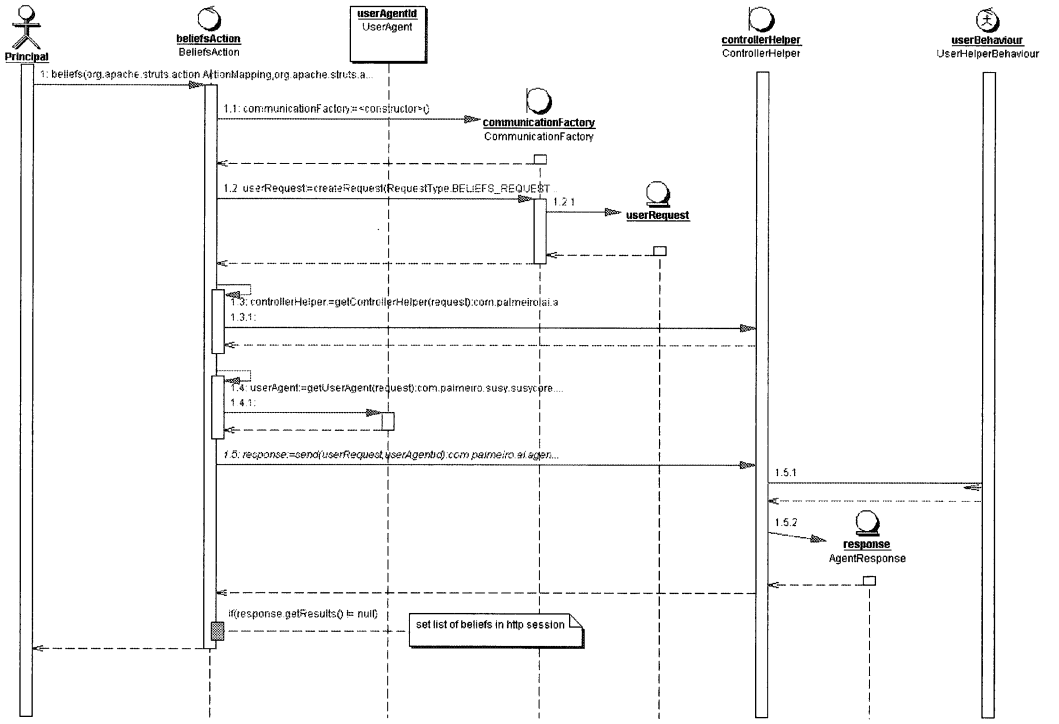

Figura 6.29: Listagem das Crenças do UA (1) - Diagrama de Sequência.

acordo com este novo "estímulo". Note-se que a base de conhecimento do agente contém regras específicas para o tratamento de pedidos do utilizador, introduzindo novas regras e afirmações (intenções) no novo estado de forma a serem tratadas na fase seguinte.

O comportamento do UA depois de adicionar o pedido do utilizador, recupera as intenções associadas (através do método *getIntentions* que por sua vez cria uma regra JLups para recuperar este tipo de dados) ao presente estado da base de conhecimento. Cada resultado recuperado (neste caso apenas um) do JLups será associado a uma intenção (objecto *Intention*) que será armazenada numa lista e passada para um método - deste comportamento - especializado no tratamento das intenções [49].

A intenção para listagem de crenças é devolvida pelo JLups com base nas regras existentes. Esta intenção é novamente traduzida para uma regra JLups responsável por uma *query* (comando *holds*) à base de conhecimento de forma a recuperar todas as crenças existentes. Do resultado desta *query* virá um conjunto de resultados que serão transformados em objectos representantes de crenças (objecto *Belief*). Esta lista de crenças será enviada numa mensagem ACL para o *ControllerHelper* que por sua vez passará o controlo novamente para a acção Struts com estes resultados. O fluxo termina com o encaminhamento para uma JSP, por parte da acção, onde serão listadas todas as crenças através da biblioteca *DisplayTag*. Como nota final, é possível ao utilizador ordenar todos os atributos de uma crença, assim como paginar todas as crenças de forma assíncrona através de tecnologia AJAX (consultar secção 6.4.3.4).

O presente fluxo encontra-se descrito nas figuras 6.29 e 6.30.

Note-se ainda que todos os fluxos que envolvem o UA e tratamento de crenças, são

[49] Note-se que esta funcionalidade poderia ficar separada do comportamento principal, como p.e. num componente dedicado apenas a esta causa. Para efeito de simplicidade inicial esta funcionalidade foi inserida no comportamento do agente, no entanto com o crescimento das potencialidades do agente e da complexidade associada às sua regras está prevista a construção de um componente dedicado ao tratamento de intenções.

idênticos ao presente apenas alterando a forma de tratamento de cada intenção (método *handleIntention*, ver fluxo 2.4.2.1 do diagrama de sequência da figura 6.30). Da mesma forma a adição e remoção de crenças, assim como a pesquisa de dados juntamente com apoio gramatical são fluxos totalmente idênticos, não sendo portanto descritos como cenários mais significativos.

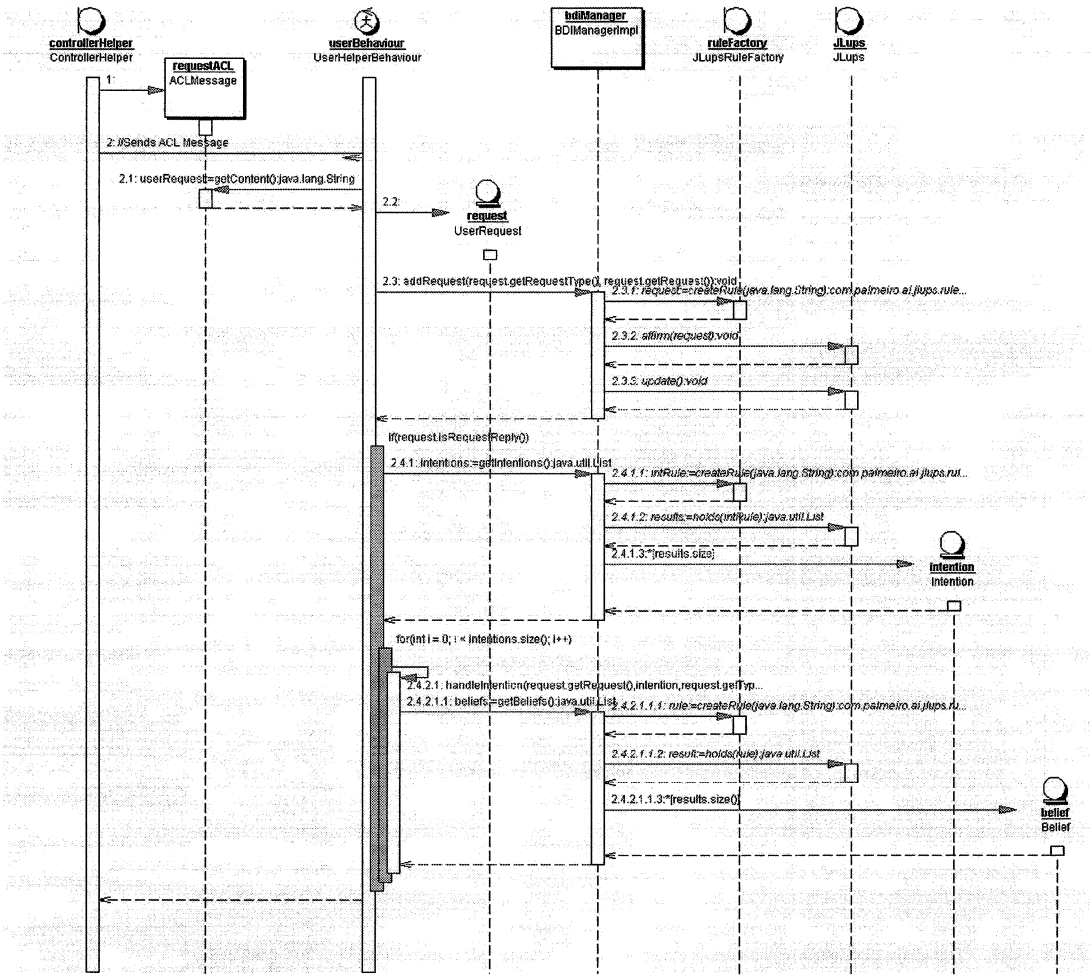

Figura 6.30: Listagem das Crenças do UA (2) - Diagrama de Sequência.

6.4.7.4 Pesquisa Interactiva, Colaboração e Apoio à Recuperação de Informação

Descrição

Esta secção aborda o processo de pesquisa de informação e respectivas operações na camada *model*. Este fluxo é considerado significativo para a arquitectura pois descreve um exemplo de cooperação entre agentes de serviços, juntamente com o agente do utilizador. Também explora o uso de tecnologias assíncronas como forma de auxiliar a recuperação de informação.

Os três tipos de pesquisas existentes no sistema são a pesquisa simples, automatizada e interactiva. Estas pesquisas fazem parte do Caso de Uso "UC2 Pesquisar"e estão descritas no Capítulo 5 Análise Funcional, secção 5.2.2.

É importante notar que apesar do Capítulo 5 descrever os três fluxos de pesquisas

através de dois "*workers*", ao nível arquitectural encontram-se três "*workers*": o SA e o GA (já existentes, ver figuras 5.5, 5.6 e 5.7) e o UA, onde fica armazenado todo o conhecimento e toda a lógica que define os fluxos. O SA e GA são agentes que fornecem serviços e são cegos perante a lógica do sistema.

Do ponto de vista arquitectural foi escolhido o cenário da pesquisa interactiva, de forma a exemplificar a modelação ao nível da base de conhecimento do UA, e demonstrar os fluxos cooperativos entre agentes (UA, SA e GA).

Por se usar a mesma heurística para todo o tipo de comunicação entre PA e UA através do componente *ControllerHelper*, a descrição deste sub-fluxo será omitida na presente funcionalidade[50].

Estrutura

A estrutura do presente fluxo é constituída por vinte e três classes e sete interfaces. Note-se que existem mais classes envolvidas, no entanto as descritas apresentam um carácter mais significativo (ver figura 6.31 e 6.32):

- *AgentResponse*;
- *AgentResponseImpl*;
- *AjaxXmlBuilder*;
- *BaseAjaxAction*;
- *BeliefsAutocompleteAction*;
- *BDIManager*;
- *BDIManagerImpl*;
- *CommunicationFactory*;
- *ControllerHandlerBehaviour*;
- *ControllerHelper*;
- *ControllerHelperImpl*;
- *GoogleSearchEngine*;
- *GrammarAgent*;
- *GrammarManager*;
- *GrammaticalBehaviour*;
- *HeartBeatBehaviour*;
- *Intention*;
- *Item*;
- *JLups*;
- *JLupsImpl*;

[50]Para mais detalhes consultar os cenários descritos na secção anterior (6.4.7.3).

- *JWordNetManager*;
- *OldRequestsHandlerBehaviour* ;
- *ProxyAgent*;
- *SearchAgent*;
- *SearchBehaviour*;
- *SearchManager*;
- *UserAgent*;
- *UserHelperBehaviour*;
- *UserRequest*;
- *UserRequestImpl.*

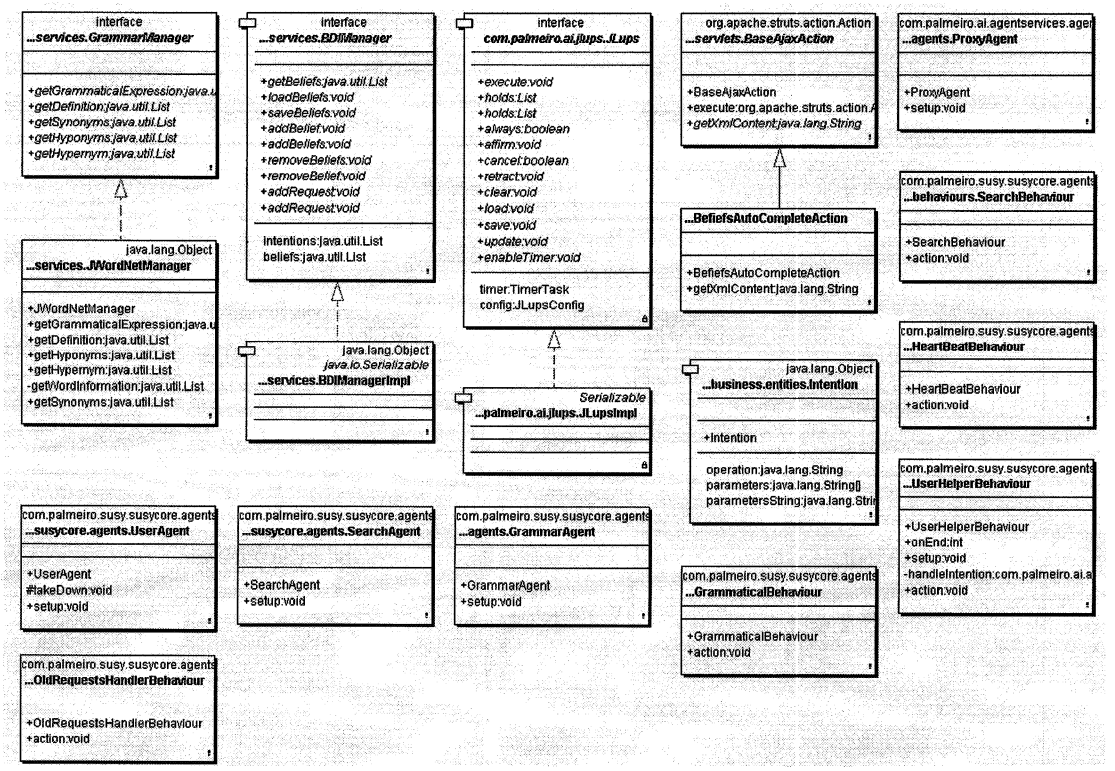

Figura 6.31: Pesquisa Interactiva (1) - Diagrama de Classes.

Cenários

Pesquisa Interactiva, Colaboração entre UA, SA e PA: Uma pesquisa interactiva exige uma colaboração entre o utilizador e o sistema, dividindo-se em dois passos principais. No primeiro o sistema é encarregue de fornecer ao utilizador - via JSP - relações gramaticais sobre os termos da pesquisa. O utilizador deve posteriormente seleccionar quais as relações interessantes[51], e com base nestas o sistema realizará novas

[51] Através de *checkboxes*.

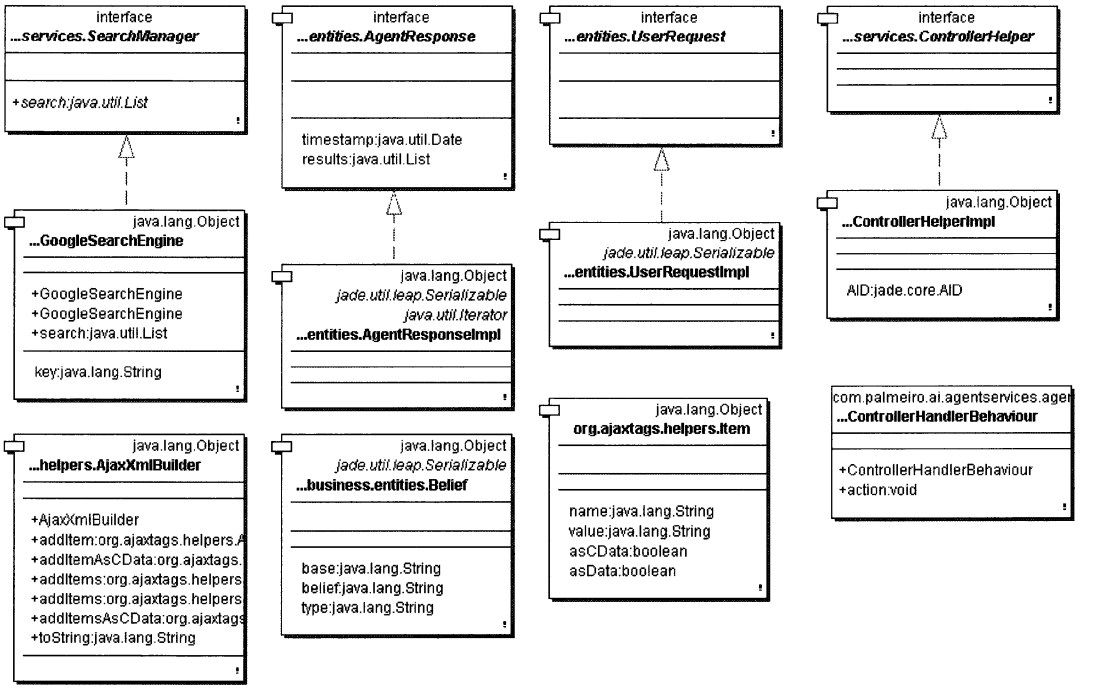

Figura 6.32: Pesquisa Interactiva (2) - Diagrama de Classes.

pesquisas. Todas as relações seleccionadas pelo utilizador serão armazenadas para uso posterior.

Neste fluxo existe uma colaboração entre diversos agentes, nomeadamente o PA, UA, SA e GA, cada um com quota parte de responsabilidade no sucesso do presente cenário. O detalhe deste fluxo será descrito através de um exemplo abstracto de uma pesquisa, acompanhado com a descrição das regras JLups usadas no processo. O diagrama de colaboração da figura 6.33 apresenta a ordem dos fluxos que são descritos nos parágrafos que se seguem.

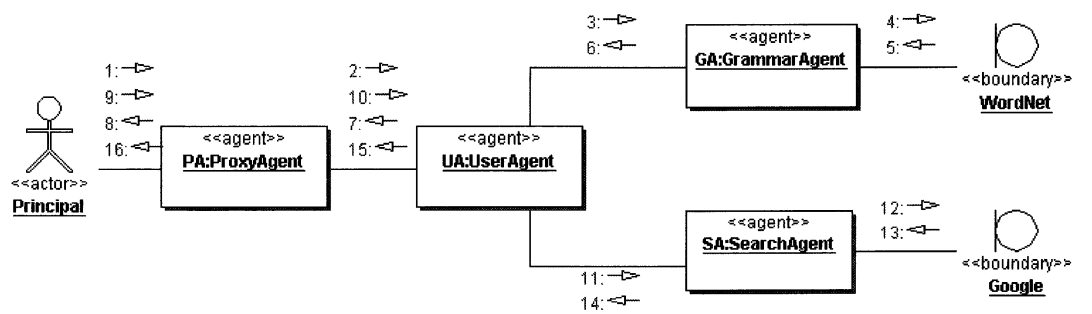

Figura 6.33: Pesquisa Interactiva - Diagrama de Colaboração.

O UA ao receber do PA um pedido do utilizador (p.e. o utilizador escreve "star" na página de pesquisa) para uma pesquisa interactiva, vai adiciona-lo à sua base de conhecimento através da invocação do seguinte método:

$$bdiManager.addRequest(interactive1, "star"); \quad (6.1)$$

A invocação deste método traduz-se na seguinte regra:

$$assert\ (request(interactive1, "star")) \tag{6.2}$$

Note-se que o agente nunca desce ao nível das regras JLups. Este limita-se apenas usar uma interface de alto nível para trabalhar com a sua base de conhecimento. Este comando, tal como os restantes, estão encapsulados no serviço *BDIManager*.

O UA por cada interacção - finalizada com uma actualização através do método *update*[52] - recupera as intenções para o presente estado, através da invocação do seguinte método:

$$bdiManager.getIntentions(); \tag{6.3}$$

A invocação deste método traduz-se na seguinte *query*:

$$holds\ int(X, Y) \tag{6.4}$$

A primeira intenção é recebida através da regra 6.5 (que faz parte do conjunto inicial de regras de um UA), em que por cada pedido interactivo inserido pelo utilizador o UA requer suporte ao GA:

$$always\ event\ (int(ga, R))\ when\ (request(interactive1, R)) \tag{6.5}$$

Como resultado desta intenção o UA envia uma mensagem ao GA de forma a que este envie uma lista de relações gramaticais para a palavra "star". O GA usa o serviço *GrammarManager* para recuperar esta relação gramatical do WordNet, enviando uma resposta ao UA com uma lista de resultados. O UA recebe esta lista, envia para o PA residente no componente *ControllerHelper* e a acção de crenças retoma o controlo, mostrando a lista de sinónimos/hipónimos/hiperónimos ao utilizador. O utilizador selecciona quais as sugestões que lhe poderão interessar, como p.e. "supernova", e prossegue com a pesquisa passando o controlo para a acção onde o PA enviará ao UA as opções do utilizador. O UA ao receber esta mensagem cria uma crença entre as palavras "star" e "supernova", através do método para adicionar uma lista de crenças:

$$bdiManager.addBeliefs(beliefs); \tag{6.6}$$

No exemplo em causa a lista *beliefs* apenas contém uma crença encapsulada num objecto *Belief*. Ao nível do JLups uma crença é armazenada através do predicado *belief('gramatical association', 'base concept', 'second concept')*.

A invocação deste método traduz-se na introdução de uma crença com a seguinte forma:

$$assert\ (bel("synonym", "star", "supernova")) \tag{6.7}$$

Logo após a adição de novas crenças o UA inicia a segunda fase do fluxo, inserindo o pedido para pesquisa por parte do utilizador:

$$bdiManager.addRequest(interactive2, "star"); \tag{6.8}$$

Neste momento e após um *update*, o UA contém crenças associadas à palavra "star" e através da seguinte regra vai refinar a informação:

[52] Após a inserção de cada "afirmação" o UA actualiza sempre a base de conhecimento, de forma a que as regras existentes façam efeito. Doravante não se irá referir este facto, devendo ser considerado como implicíto.

$$always\ event\ (refine(R))\ when\ (request(interactive2, R)) \tag{6.9}$$

Para refinar a informação, usam-se as seguintes regras que vão ser responsáveis pela criação de uma intenção por cada relação gramatical com a palavra "star".

$$always\ event\ (int(search, R + S))\ when\ (bel(G, S, R), refine(R)) \tag{6.10}$$

$$always\ event\ (int(search, R + S))\ when\ (bel(G, R, S), refine(R)) \tag{6.11}$$

As intenções neste caso são pedidos de pesquisas ao SA. O UA através da regra 6.4 detecta uma intenção e envia para o SA um pedido de pesquisa com o conteúdo "star supernova". O SA recebe o pedido, usa o *web service* da Google como forma de pesquisa e retorna uma lista de resultados ao UA, que por sua vez os redirecciona para o PA. O controlo volta novamente à acção de crenças onde serão finalmente apresentados os resultados ao utilizador.

Continuação, Pesquisa Automatizada: Continuando o fluxo anterior, este cenário descreve de forma simplificada as regras que seriam usadas para uma pesquisa automatizada, de forma a que o leitor retenha as diferenças existentes entre os dois fluxos de pesquisa.

Após a primeira pesquisa (cenário anterior) o utilizador usa nesta fase a pesquisa automatizada. De forma análoga à regra 6.2, o UA adiciona o pedido de pesquisa sobre "supernova"e actualiza a base de conhecimento:

$$assert\ (request(automated, "supernova")) \tag{6.12}$$

Uma pesquisa automatizada tem alguns fluxos internos que são descritos com as seguintes regras:

$$always\ event\ (request(simple, R) \leftarrow request(automated, R)) \tag{6.13}$$

$$always\ event\ (int(search, R))\ when\ (request(simple, R)) \tag{6.14}$$

A regra 6.13 indica que para cada pedido automatizado irá existir um pedido de pesquisa simples. Por sua vez a regra 6.14 mapeia um pedido de pesquisa simples para uma intenção de pesquisa.

$$always\ event\ (refine(R))\ when\ (request(automated, R)) \tag{6.15}$$

A regra 6.15 indica que um pedido de pesquisa automatizada irá dar origem a uma "vontade" de refinar o conteúdo deste pedido. Através das regras 6.10 e 6.11 esta vontade transforma-se numa intenção de pesquisa do conteúdo juntamente com as associações gramaticais existentes.

A regra 6.16 é usada quando a prova das condições das regras 6.10 e 6.11 falha (não sendo o caso), por não existirem crenças associadas ao termo de pesquisa. Neste caso todas a relações fornecidas pelo GA seriam automaticamente armazenadas como crenças. A árvore de derivação deste fluxo terminava com a pesquisa de todas a crenças recentemente adquiridas.

$$always\ event\ (int(gramm, R))\ when\ (request(automated, R), \\ not\ bel(synonym, R, S)) \tag{6.16}$$

O UA após inserir o pedido 6.12 e actualizar a sua base de conhecimento, irá deparar com duas intenções de pesquisa: a primeira é uma pesquisa simples derivada das regras 6.13 e 6.14, enquanto a segunda intenção é uma pesquisa com o seguinte conteúdo "supernova star", derivada das regras 6.15 e 6.10 usando conhecimento adquirido em interacções anteriores.

Este fluxo, embora simples, demonstra de forma clara como uma solução DLP pode ser usada em agentes de forma a fornecer-lhes raciocínio, aprendizagem e cooperação.

Apoio ao Retorno de Informação com AJAX: À parte da funcionalidade de pesquisa através da interacção entre PA, UA, SA e GA, a plataforma fornece ainda uma interessante característica de apoio na recuperação de informação através de uma funcionalidade de "*autocomplete*".

Aquando da introdução dos parâmetros no *form* de pesquisa, por parte do utilizador, o sistema irá detectar a inserção de caracteres e ao mesmo tempo consultar as crenças existentes no agente. Se os caracteres até ao momento introduzidos pertencerem aos caracteres de uma possível crença, o sistema irá sugerir a crença assim como as suas relações já existentes como *autocomplete*. Desta forma o utilizador pode em tempo real conhecer as crenças relacionadas com o seu pedido, e pode automaticamente refinar a informação consoante os seus desejos ou usar a informação armazenada pelo seu agente em interacções anteriores.

Esta funcionalidade demonstra como uma solução com a camada *model* orientada a agentes pode trazer benefícios a diversos tipos de ambientes, neste caso um contexto de recuperação de informação. O uso desta tecnologia com um forte enriquecimento na camada de apresentação, tornam este fluxo bastante interessante e completo do ponto de vista tecnológico.

Tecnicamente o presente fluxo usa tecnologias assíncronas para a realização desta funcionalidade. É usada a tecnologia AJAX para retirar os caracteres a serem escritos pelo utilizador, para o posterior pedido assíncrono ao servidor com os parâmetros presentes no *form*, e para a apresentação de resultados vindos do servidor em forma de *autocomplete*.

Por cada palavra que o utilizador insere no *form*, uma função em javascript vai consultar via HTPP uma acção Struts (ver figura 6.34). Esta acção, que extende uma acção fornecida pela biblioteca AjaxTags, devolve conteúdo XML em cada invocação. Como primeiro passo irá consultar as crenças do utilizador que fazem *match* com os caracteres até agora inseridos, ordená-las e devolve-las em formato XML através da classe utilitária *AjaxXmlBuilder*.

Como uma solução deste tipo exige um elevado desempenho por apresentar uma forte interacção com o utilizador, a presente implementação é puramente orientada ao desempenho. Usam-se portanto as crenças que foram armazenadas em sessão, depois de uma consulta de crenças por parte do utilizador. Desta forma evita-se um pedido ao UA por uma listagem de crenças, que poderia demorar mais que o tempo necessário

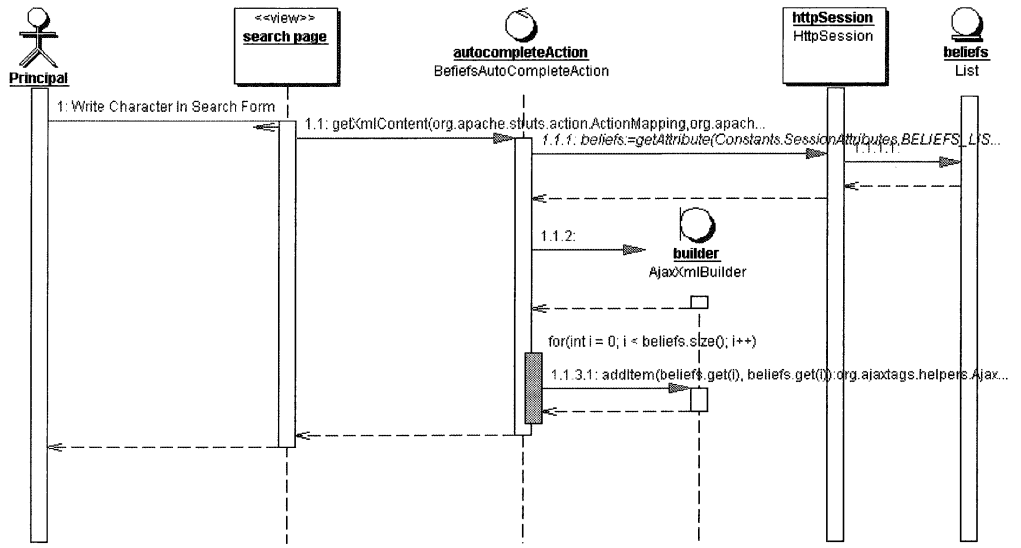

Figura 6.34: Apoio ao Retorno de Informação com AJAX - Diagrama de Sequência.

para uma funcionalidade deste tipo ser útil. Uma vez recebida a mensagem por parte do *browser*, ir-se-á lançar um lista de possíveis entradas em forma de *autocomplete*.

Note-se a simplicidade com que se pode enriquecer a interface *web* com uso de funcionalidades assíncronas, através da interacção com a camada *model* seja ela orientada a agentes ou a objectos. De igual forma poder-se-ia ter consultado o UA, ou qualquer outro agente presente no sistema.

6.5 Vista de Execução

6.5.1 Processos de Negócio

6.5.1.1 Nanny Agent - Heartbeat e Controlo do Sistema

No MAS da plataforma Susy existem vários serviços disponibilizados por agentes especializados, como por exemplo o agente de pesquisas (SA), o agente de sumarização (DSA) e o agente de apoio gramatical (GA). Todos estes agentes fornecem serviços a agentes clientes, normalmente agentes dos utilizadores. De forma a manter a estabilidade do sistema e preservar a qualidade de serviço, estes agentes são controlados de forma próxima por um outro agente presente no sistema, o *Nanny Agent* (NA).

O NA apresenta como única responsabilidade a monitorização do ciclo de vida dos agentes de serviços, estando estes num *container* local ou remoto ao NA. A monitorização é realizada através de um *heartbeat* enviado por cada serviço quando periodicamente estimulado pelo NA.

Como descrito no diagrama de sequência da figura 6.35, o NA é iniciado com uma lista de serviços existentes no sistema. O comportamento base do NA (*NannyBehaviour*) extende a classe *TickerBehaviour*[53], herdando a funcionalidade de executar periodicamente um bloco de código, encapsulado no método *onTick*. O período definido para execução deste código está definido no *deployment descriptor* (web.xml) da aplicação web, sendo recuperado por uma *servlet* de inicialização do sistema que lança todos os agentes presentes no MAS (excepto os UAs e PAs).

[53] jade.core.behaviours.TickerBehaviour

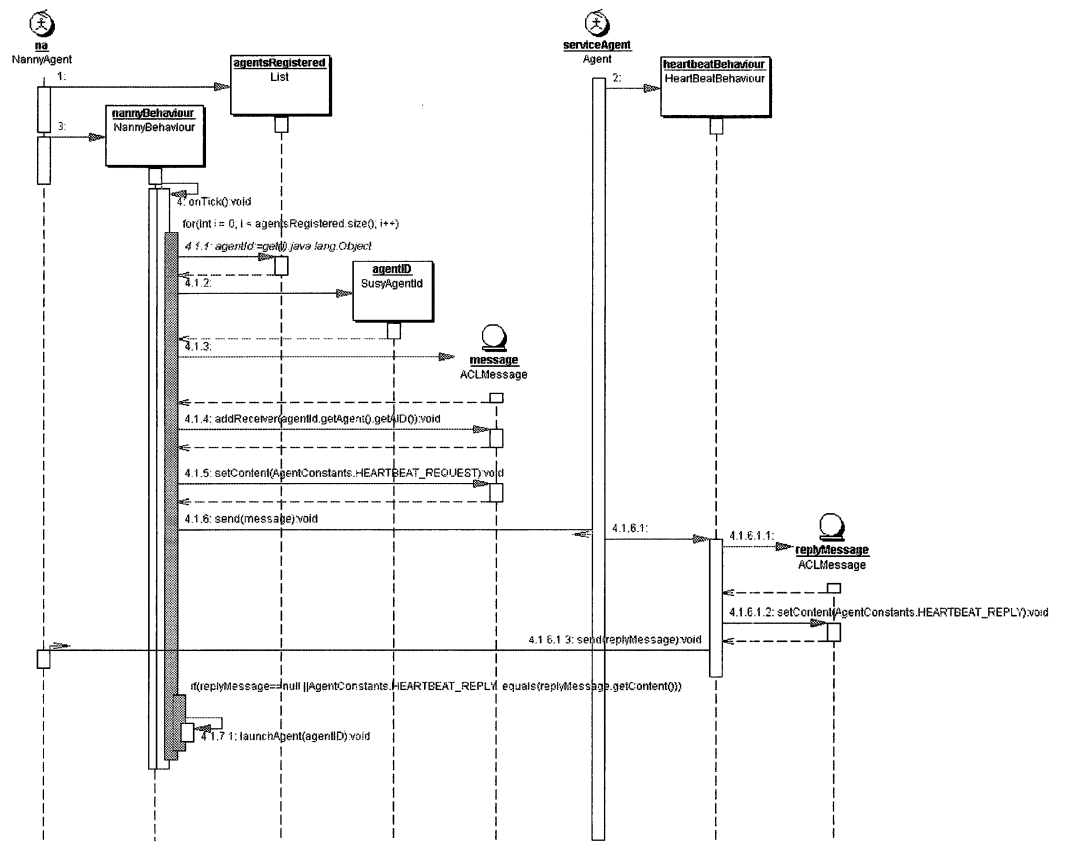

Figura 6.35: Nanny Agent e o comportamento *Hearbeat* - Diagrama de Sequência.

Por cada *tick* existente no sistema o comportamento principal do NA vai iterar uma lista de serviços, enviando para cada um uma mensagem ACL contendo um pedido de sinal de vida. O serviço destino contendo o comportamento *HeartbeatBehaviour*[54] irá responder com uma mensagem tipo "*I'm alive*" sinalizando desta forma o seu *heartbeat*. Se o NA não receber resposta num determinado espaço temporal, irá tentar relançar o serviço em possível falha. Caso o serviço esteja funcional e o atraso da mensagem de resposta seja devido a taxas de ocupação elevadas, o relançamento irá falhar e o NA irá assumir que se tratou de um "mau julgamento" de sua parte. O NA funciona portanto com um algoritmo pessimista.

Note-se que o NA nesta versão apenas possuí regras simples, sendo que num futuro este agente poderá inclusive ter mais poder tanto na criação como na gestão de agentes de serviços.

6.5.2 Processos de Sistema

6.5.2.1 Inicialização da Plataforma

O sistema Susy, que incluí um conjunto alargado de estilos arquitecturais, é inicializado através da *servlet SusyConfigServlet*. Esta *servlet* é responsável pelo lançamento do MAS, assim como da recuperação de todos os parâmetros necessários para o funcionamento dos restantes subsistemas (JLups, Configuração de agentes, entre outros). Este fluxo é apenas realizado uma única vez e durante a inicialização do servidor aplicacional.

[54]Ver secção 6.4.5.2 Agentes de Serviços nas Regras de Desenho.

A inicialização da plataforma divide-se portanto em três fases principais (ver figura 6.36):

1. Recuperação dos Parâmetros de Configuração;
2. Inicialização da Plataforma de Agentes;
3. Lançamento dos Agentes Vitais para a Plataforma.

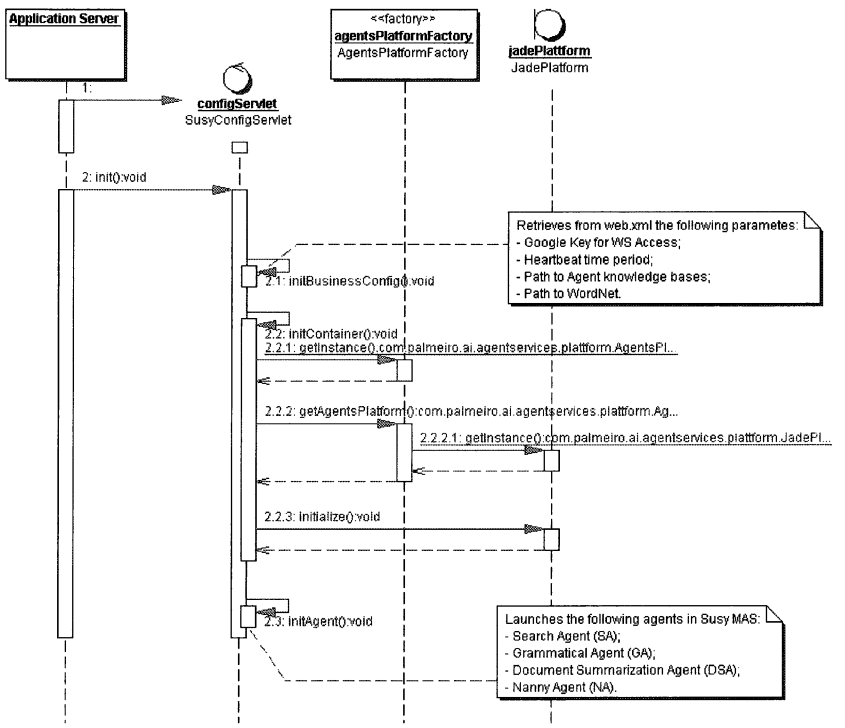

Figura 6.36: Lançamento da Plataforma Susy - Diagrama de Sequência.

Na primeira fase realiza-se uma recuperação de vários parâmetros de configuração essenciais para os subsistemas presentes na plataforma. Estes parâmetros estão marcados no *deployment descriptor* (*web.xml*) da aplicação *web* (susyweb). Todos estes parâmetros irão ser armazenados num objecto de negócio único no sistema (*BusinessConfig*, que implementa a *pattern singleton*, ver secção 6.4.6.2 Camada Específica ao Domínio), que através duma tabela de dispersão mantém entradas "parâmetros/valores"a serem usadas por todos os subsistemas. De entre estes parâmetros são de destacar:

- Chave de Acesso ao Google: Esta chave é usada pelo Agente de Pesquisas (SA) e é necessária para aceder ao *web service* disponibilizado pela Google;
- Período de *Heartbeat*: Período que o *Nanny Agent* (NA) aguarda até à próxima verificação do estado dos serviços;
- Caminho de Acesso às Bases de Conhecimento dos Agentes: Caminho no sistema de ficheiros para onde estão armazenadas as bases de conhecimento dos agentes em formato JLups (ficheiros com extensões *.xml* e *.p*). Usado por todos os agentes do utilizador no sistema;

- Caminho Para o WordNet: Caminho no sistema de ficheiros para onde está a biblioteca WordNet instalada. Este parâmetro é usado pelo Agente Gramatical (GA).

Após o carregamento inicial dos parâmetros de configuração é iniciada a segunda fase, que corresponde à inicialização do MAS. Note-se na figura 6.36 como este processo é desencadeado. A *servlet* de configuração usa uma *factory* (*AgentsPlatformFactory*) responsável pela criação de plataforma de agentes, ocultando a implementação usada e fornecendo apenas acesso ao MAS através da interface definida (*AgentsPlatform*, consultar secção 6.4.6.3 *Middleware* de Agentes - Módulo AgentServices). Após a plataforma ser criada, a *servlet* de configuração invoca o método *initialize* de forma a dar início ao lançamento da plataforma de agentes, neste caso a plataforma JADE.

Uma vez iniciada a plataforma e tendo todos os parâmetros encapsulados num objecto de negócio, esta *servlet* vai lançar todos os agentes principais para a plataforma:

- Agente Nanny (NA);
- Agente de Pesquisa (SA);
- Agente de Apoio Gramatical (GA);
- Agente Sumarizador de Documentos (DSA).

Para mais detalhes sobre os agentes supracitados consultar secção 6.4.1.2 Camada de Serviços de Negócio.

É importante notar que qualquer erro ocorrido nesta *servlet* será devidamente reportado para os *logs* do servidor, tendo como resultado a finalização do fluxo e o não lançamento da plataforma Susy. É portanto necessário que todos os parâmetros estejam bem configurados, assim como é necessário estarem disponíveis todos os subsistemas e respectivas bibliotecas.

6.6 Vista de Instalação

Esta secção descreve o modo como os componentes serão distribuídos em ambiente de execução para cada uma das configurações identificadas na arquitectura de sistemas.

6.6.1 Estrutura de Componentes

A figura 6.37 apresenta os diversos componentes que compõem a plataforma Susy. Para mais detalhe consultar secção 6.4 Vista Lógica.

6.6.2 Distribuição em Ambiente de Execução

A distribuição dos diversos componentes da plataforma Susy em ambiente de execução segue a estrutura de *deployment* apresentada na figura 6.38.

Note-se que o *container* de agentes é lançado aquando da inicialização da plataforma, através do processo descrito na secção 6.5.2.1 Inicialização da Plataforma. No entanto agentes clientes podem residir num *container* remoto e interagir de igual forma com a plataforma. Por exemplo, todos os serviços disponíveis (SA, DSA e GA) poderiam fornecer as suas funcionalidades remotamente, preservando e fortalecendo o carácter distribuído da solução.

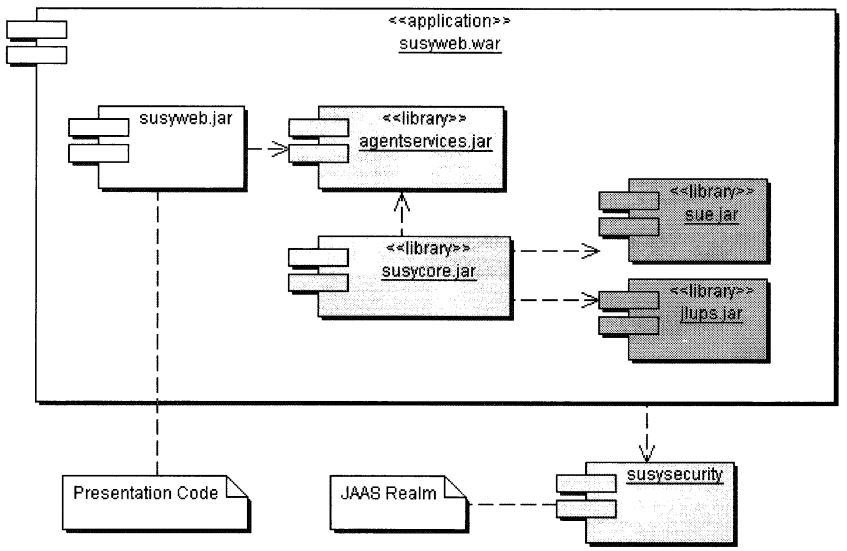

Figura 6.37: Plataforma Susy - Diagrama de Componentes.

6.6.3 Regras para Instalação

Nesta secção resumem-se a regras de instalação para Tomcat 5.5. De forma a preparar o ficheiro *war* é necessário lançar o *script* Maven[55] ("maven susy") para construir a aplicação *web*. Todas as dependências encontram-se parametrizadas na "configuração Maven" de cada módulo presente no sistema.

1. De forma a que a autorização esteja disponível torna-se necessário instalar a biblioteca susysecurity (disponibilizada no ficheiro susysecurity-1.0-rc1.jar) e o módulo JAAS (tagishauth.jar) na pasta TOMCAT/server/lib;
2. Definir o *realm* no ficheiro server.xml através da seguinte entrada:

```
<Realm className="com.palmeiro.susy.security.TomcatRealm"
       debug="999" appName="FileLogin"
           userClassNames="com.tagish.auth.TypedPrincipal"/>
```

3. Compilar e criar um ficheiro *war* através dos *scripts* Maven, usando o comando "maven susy". O resultado deste processo é o ficheiro *susy.war*;
4. Instalar o ficheiro *susy.war* no servidor através da interface *web* ou de forma manual;
5. Configurar os parâmetros presentes da *servlet ConfigServlet* no ficheiro *web.xml*;
6. Depois de se lançar o servidor a plataforma Susy estará acessível através de um URL do tipo "http://host:port/susy". Actualmente o acesso ao ambiente de qualidade é realizado através do URL "http://alinex-3.di.uevora.pt:8080/susy".

Pode-se consultar os ficheiros de *log* gerados pelo servidor, de forma a se conhecer o estado da plataforma Susy depois do processo inicialização. Qualquer erro existente neste processo (*ConfigServlet*) será descrito nestes *logs*, assim como qualquer erro derivado de configurações externas erradas.

[55] Ver http://maven.apache.org.

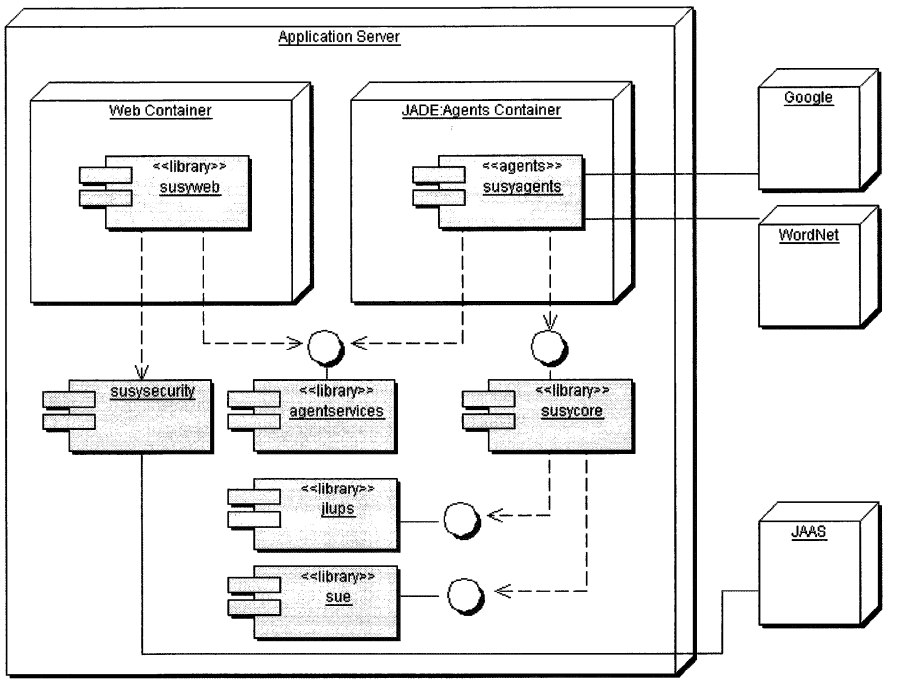

Figura 6.38: Plataforma Susy - Diagrama de *Deployment.*

6.7 Vista de Implementação

Esta secção descreve a organização para os módulos de software no ambiente de desenvolvimento.

6.7.1 Organização do Código na Árvore de Projecto

A árvore de projecto encontra-se descrita na tabela 6.1. Cada entrada nas pastas "apps" e "components" corresponde a um módulo em *Concurrent Version System* (CVS). No topo da hierarquia existe o módulo CVS "susy" que os engloba a todos. Como projectos independentes surgem o JLups e Sue, estando este último fora do âmbito da presente dissertação.

/susy			
	/apps		
		/susyweb	[Applicação Web - WAR]
	/components		
		/agentservices	[*Middleware* de Agentes - JAR]
		/susycore	[Núcleo, Estruturas e Fundações - JAR]
		/susysecurity	[Módulo de Segurança - JAR]
/jlups			[Componente DLP - JAR]
/sue			[Componente de Sumarização - JAR]

Tabela 6.1: Árvore de Projecto.

6.7.2 Estrutura de *Packages*

A estrutura dos *packages* segue a estrutura de *layers* e subsistemas identificados no presente documento. O código ficará residente nessas pastas, em *project/src*, de acordo com as seguintes convenções para os nomes de *packages* de implementação:

- *Package* base do projecto : *com.palmeiro.susy*
- Camada de Apresentação : *com.palmeiro.susy.susyweb.presentation.[controller — validation]*
- Camada de Negócio : *com.palmeiro.susy.susycore.business.[entities — services — agents]*
- Camada de Segurança : *com.palmeiro.susy.susysecurity*

Os componente com maior nível de independência como o *AgentServices* e JLups seguem as seguintes convenções:

- *Package* base do projecto : *com.palmeiro.ai.[jlups — agentservices]*

A sub-estrutura do componente JLups encontra-se detalhada no Capítulo 7, enquanto o componente *AgentServices* segue uma sub-estrutura idêntica às camadas acima definidas[56].

As classes de testes unitários estão localizadas na pasta */test* da raíz de cada projecto, no mesmo *package* da classe que testam.

6.7.3 Estrutura da Aplicação *Web*

A estrutura seguida para armazenar a aplicação *web* desenvolvida no âmbito da presente tese, segue a árvore definida na tabela 6.2.

/susyweb			
	/src		[Código da camada de apresentação]
	/model		[Modelação da Aplicação]
	/web		[Conteúdo do WAR]
		/images	[Imagens]
		/js	[Javascript]
		/styles	[CSS]
		/WEB-INF	[web.xml, dwr.xml, jsp, etc]
		agents	[Bases de Conhecimentos dos UAs]
		tld	[Taglibraries]
		security	[Configurações JAAS]
		jsp	[JSPs]
		conf	[Struts, Tiles e Validations]

Tabela 6.2: Estrutura de Webapps.

6.7.4 Repositório de Bibliotecas

A plataforma Susy devido à sua abrangência arquitectural usa um conjunto alargado de bibliotecas. A tabela 6.3 resume este conjunto de dependências assim como as suas respectivas versões.

6.7.5 Regras de Implementação

Devem ser seguidos os *coding standards* definidos pela Sun, não esquecendo que todas as classes deverão ter um cabeçalho GNU com a identificação do(s) autor(es) e o número da revisão no CVS [57]. Todos os métodos públicos e protegidos deverão ter *javadocs* associados.

[56] Para mais detalhes consultar secção 6.4.6.3 *Middleware* de Agentes - Módulo AgentServices.

[57] Um exemplo é apresentado no Capítulo 7, secção 7.6.2.

Biblioteca	Versão
dwr	1.1
jericho-html	2.1
mandarax	3.4
log4j	1.2.8
jade	3.3
commons-beanutils	1.6
commons-collections	3.1
commons-digester	1.7
commons-lang	2.1
commons-logging	2.0.1
commons-validator	1.1.4
displaytag	1.0
oro	2.0.8
jstl	1.1.2
standard	1.1.2
antlr	2.7.2
servletapi	2.4
ajaxtags	1.2-beta1
struts	1.2.8
susycore	1.0-rc1
susysecurity	1.0-rc1
agentservices	1.0-rc1
jlups	1.0-rc09
sue	1.1
googleapi	n/a
jwordnet	n/a

Tabela 6.3: Bibliotecas e versões usadas na plataforma Susy.

Podem ser criadas novas classes ou componentes, sendo no entanto necessário garantir que seguem as boas prática e regras descritas ao longo desta dissertação (com foco na secção 6.4 Vista lógica), e seguem também as indicações descritas na secção 6.7.1 Organização do Código na Árvore de Projecto

Os comentários, os *javadocs*, os nomes das classes, os nomes dos métodos e o próprio código deverão estar em inglês.

Devem ser seguidas as nomenclaturas para classes, em que por exemplo as classes abstractas deverão começar por '*Abstract*', as classes que implementem *patterns* de desenho deverão terminar com o sufixo correspondente ao papel desempenhado - *Factory*, *Façade*, *Manager*, *Visitor*, *Decorator*, entre outros.

Devem ser efectuados testes unitários a cada serviço presente na camada de negócio, usando casos de teste JUnit[58]. Todas as operações para a classe em causa devem ser testadas. Deve ainda ser utilizada a API *open-source Struts-Test*[59] para testes de integração nas acções Struts.

6.8 Resumo

O Capítulo 8 resume toda a arquitectura proposta.

[58]Ver http://www.junit.org.

[59]Ver http://sourceforge.net/projects/strutstestcase.

Capítulo 7

JLups, Componente DLP

Este capítulo apresenta as vistas arquitecturais que se destinam a modelar o componente JLups. É objectivo descrever com o maior detalhe os componentes identificados, descrevendo a sua implementação e sub-componentes que os preencham.

7.1 Introdução

7.1.1 Enquadramento

O JLups é um motor de regras Java orientado para actualizações dinâmicas. Inspirado na *Language of dymamic UPdateS* (LUPS)[1], o projecto JLups pretende simular o seu comportamento num paradigma arquitectural diferente, sendo orientado para o uso empresarial.

A linguagem LUPS foi desenhada para especificar actualizações em programas lógicos - DLP. Dada uma base de conhecimento inicial, fornece uma forma de actualizá-la sequencialmente. O JLups reaproveita o conceito introduzido pela linguagem LUPS e disponibiliza-o na linguagem Java, de forma a potenciar o desenvolvimento de aplicações e programas a um nível diferente do original. Mais informações sobre DLP e LUPS podem ser encontradas no Capítulo 3 e em [APP99b], [APPQa] e [APPQb].

7.1.2 Organização

O presente capítulo segue uma organização análoga à do capítulo anterior, reformulada para descrição de componentes. À parte da secção introdutória, divide-se em cinco secções.

A primeira secção aborda a Vista Funcional do sistema. A vista funcional documenta o sistema em termos dos seus objectivos e problemas que resolve. Descreve o contexto em que o sistema vai ser usado, os problemas resolvidos, os serviços que fornece e respectivos interfaces e características qualitativas destes serviços.

A segunda secção descreve a Vista Lógica, detalhando a estrutura dos componentes usados assim como os cenários existentes no sistema.

A terceira secção introduz a Vista de Execução, que descreve a decomposição do sistema em tarefas e processos.

A quarta secção aborda a Vista de Instalação, descrevendo o sistema e a sua decomposição em componentes, como bibliotecas, *packages* e dependências.

Na quinta e última secção é apresentada a Vista de Implementação, que descreve como devem estar organizados os ficheiros de software, introduz os *coding standards*

[1] Consultar http://centria.di.fct.unl.pt/~jja/updates/

usados na implementação e em futuro desenvolvimento.

7.2 Vista Funcional

7.2.1 Interface do Sistema

Os clientes do componente JLups podem ser diversos, como por exemplo agentes, clientes Java simples, aplicações *web*, entre outros.

O JLups fornece um conjunto de funcionalidades que permitem actualizar a sua base de conhecimento, nomeadamente (ver figura 7.1):

- *Assert*, introdução de regras;
- *Always*, introdução de leis;
- *Cancel*, cancelamento de leis;
- *Retract*, remoção de regras;
- *Update*, actualização do estado actual;
- *Holds*, *query* à base de conhecimento;
- *Save*, gravar a base de conhecimento para memória persistente;
- *Load*, carregar uma base de conhecimento.

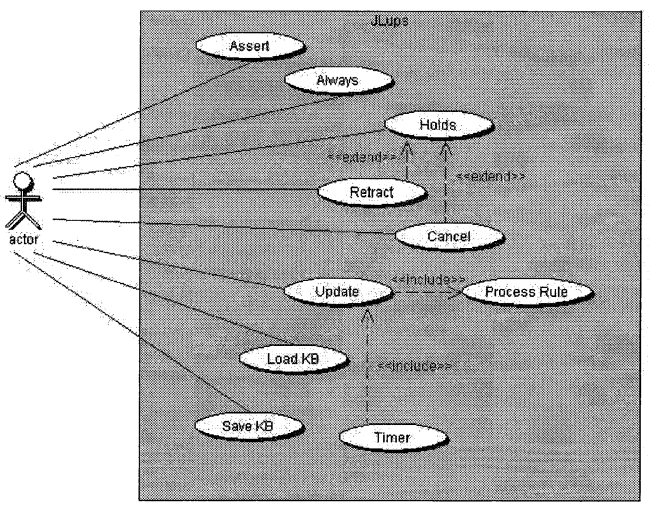

Figura 7.1: JLups - Diagrama de Casos de Uso.

Estas operações são realizadas sobre os estados em que o JLups se encontra. A cada estado, corresponde uma base de conhecimento à qual são aplicadas regras e inferências.

7.2.1.1 *Assert*

O comando *assert*, de acordo com a nomenclatura original e abaixo descrita, introduz uma nova regra.

$$assert\ [event]\ L \leftarrow L_1, ..., L_k\ [when\ L_{k+1}, ..., L_m]$$

De forma mais simples, vem:

$$assert\ [event]\ R\ [when\ C]$$

Como parâmetros opcionais, encontram-se o atributo de evento e a cláusula *when*. Se o atributo *event* for especificado, insere-se a regra no estado *t+1* (caso normal), e insere-se ainda a regra negada por defeito[2] no estado *t+2*. A cláusula *when*, se existir, ditará se a regra apresenta condições para ser inserida.

Para o efeito, considera-se que uma regra (R) se apresenta na forma de $L \leftarrow L_1, ..., L_k$. A secção antes do símbolo "$\leftarrow$" representa a conclusão, sendo a outra considerada como pré-requisitos para que a conclusão possa ser assumida. As condições para que essa regra seja válida (consequentemente inserida na base de conhecimento) surgem a seguir à cláusula *when* (C) na forma $L_{k+1}, ..., L_m$.

O processo *assert* está descrito na figura 7.2 e 7.3.

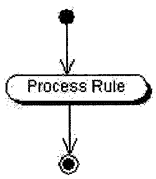

Figura 7.2: *Assert* - Diagrama de Actividade.

7.2.1.2 *Always*

Uma operação *always* introduz uma regra que vai ser aplicada em todos os estados. Este tipo de regra é usualmente chamada de lei. O comando LUPS é análogo ao *assert*.

$$always\ [event]\ L \leftarrow L_1, ..., L_k\ [when\ L_{k+1}, ..., L_m]$$

Uma regra é inserida no estado *t+1* se a condição *when* for provada e a presente regra não estiver negada por defeito. No caso de ser um evento o processo é igual ao *assert*. Como estas regras são chamadas para cada estado, guardam-se numa lista de leis.

A figura 7.4 descreve o diagrama de actividade do presente comando.

7.2.1.3 *Cancel*

Esta operação cancela uma lei introduzida com o comando *always*. Se existir uma lei idêntica, esta não será inserida para o próximo estado.

$$cancel\ L \leftarrow L_1, ..., L_k\ [when\ L_{k+1}, ..., L_m]$$

O atributo de evento não é usado nesta operação. As regras de cancelamento são armazenadas da mesmo forma que as leis, numa estrutura tipo lista.

A figura 7.5 descreve o diagrama de actividade do presente comando.

[2] A negação por defeito serve para identificar uma regra de forma a que se pare a prova para os estados anteriores.

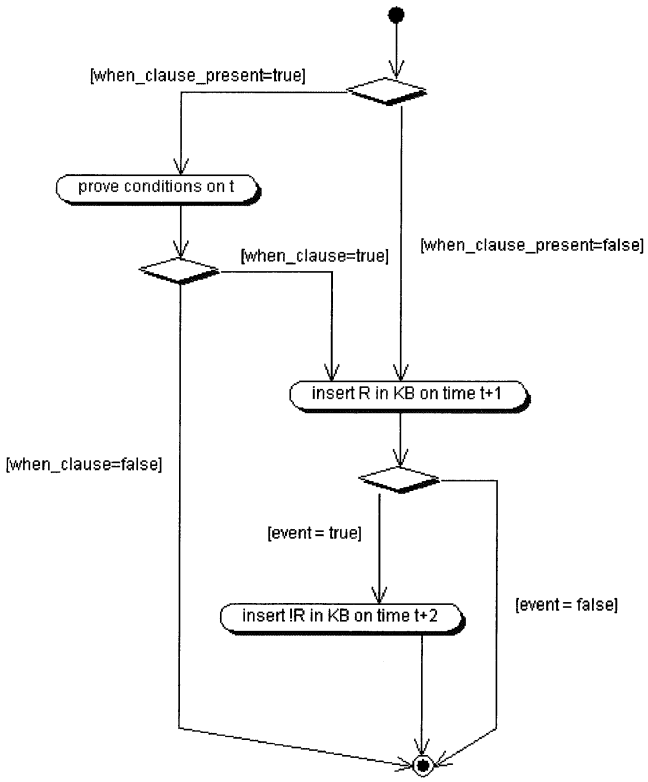

Figura 7.3: Processamento de uma regra - Diagrama de Actividade.

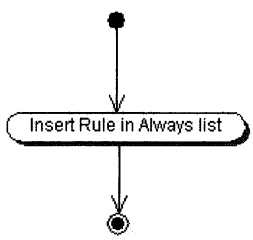

Figura 7.4: Always - Diagrama de Actividade

7.2.1.4 *Retract*

A operação de *retract* retira uma regra inserida com um *assert* da base de conhecimento JLups. Está para um *assert* como o *cancel* está para um *always*. Pode retirar-se uma regra somente uma vez usando o atributo de evento, ou retirá-la de forma permanente até uma nova introdução. A condição *when* também é usada da mesma forma que nos comandos anteriores.

$$retract\ [event]\ L \leftarrow L_1, ..., L_k\ [when\ L_{k+1}, ..., L_m]$$

A figura 7.6 descreve o diagrama de actividade do presente comando.

7.2.1.5 *Update*

Um *update* introduz um novo estado. Os estados anteriores são preservados para futuras *querys*, pois se não existirem resultados no estado actual, o JLups irá tentar provar a

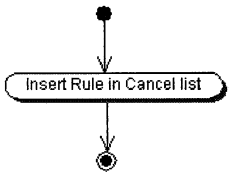

Figura 7.5: Cancel - Diagrama de Actividade

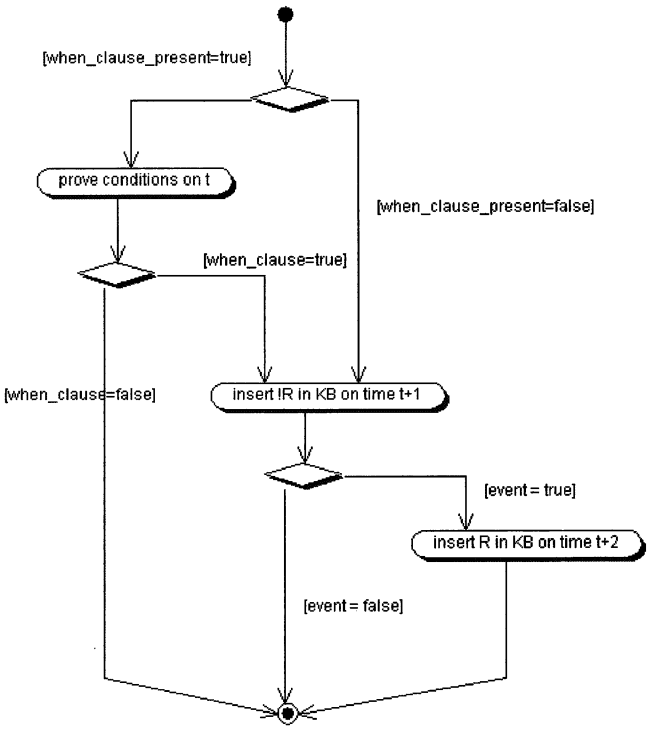

Figura 7.6: Retract - Diagrama de Actividade.

inferência para todos os estados anteriores até encontrar a prova, chegar ao estado inicial ou encontrar a negação por defeito da *query*.

Ao iniciar um novo estado, processa-se a diferença entre a lista de comandos *always* e *cancel*. O resultado desta lista, as leis que não foram canceladas, serão processadas tal como descrito na figura 7.3, e possivelmente introduzidas no novo estado.

A figura 7.7 descreve o diagrama de actividade do presente comando.

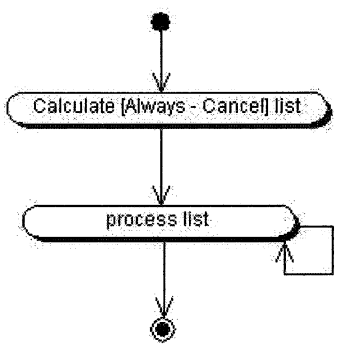

Figura 7.7: Update - Diagrama de Actividade.

7.2.1.6 *Holds*

O comando *holds* faz uma *query* ao JLups. Pode ser especificado um estado para a *query*, fazendo com que o resultado seja construído exclusivamente sobre este. No caso de ser omitido, a *query* é feita sobre todos os estados existentes.

O comando LUPS é descrito da seguinte forma:

$$holds(B_1, ..., B_k, notC_{k+1}, ..., notC_m) \ at \ q$$

ou

$$holds(B_1, ..., B_k, notC_{k+1}, ..., notC_m)$$

A figura 7.8 descreve o diagrama de actividade do presente comando.

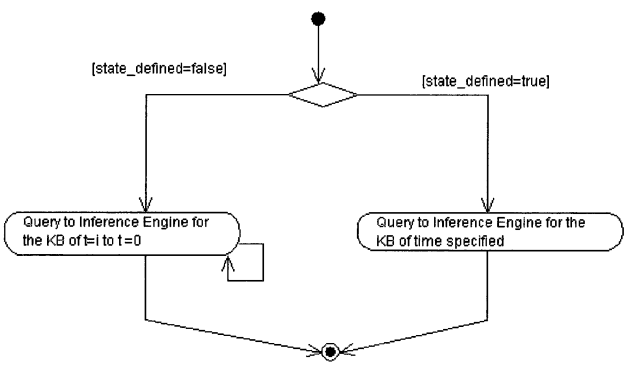

Figura 7.8: Holds - Diagrama de Actividade.

7.2.1.7 *Save*

Esta operação grava a base de conhecimento para diversos formatos, entres os quais ZKB, XKB, RULEML e formato nativo LUPS.

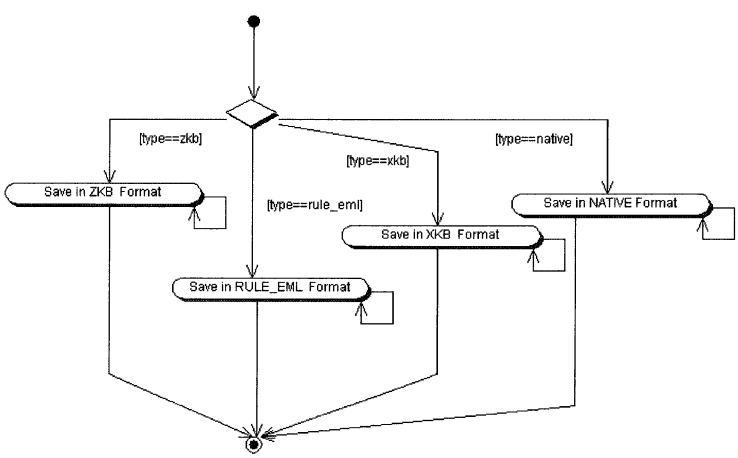

Figura 7.9: Save - Diagrama de Actividade.

7.2.1.8 *Load*

Esta operação carrega uma base de conhecimento para diversos formatos, entres os quais ZKB, XKB, RULEML e formato nativo LUPS. A descrição da base de conhecimento JLups e suas respectivas características estão gravadas num ficheiro principal.

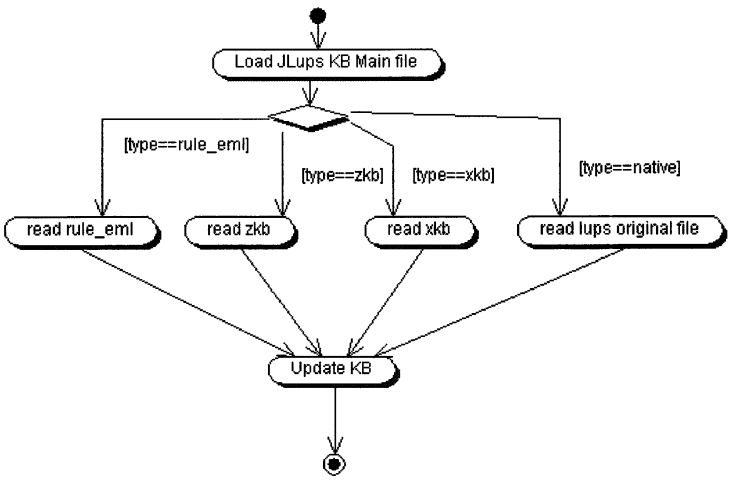

Figura 7.10: Load - Diagrama de Actividade.

7.3 Vista Lógica

A presente secção descreve os componente existentes no sistema em causa. Como o seu foco é descrever a arquitectura de um componente, usou-se como medida para esta secção as classes do componente JLups.

As funcionalidades descritas na secção 7.2 são fornecidas e incluídas nos seguintes *packages*:

- *com.palmeiro.ai.jlups*: Contém núcleo do JLups. Toda a lógica principal está implementada neste *package*;
- *com.palmeiro.ai.jlups.io*: Contém todas as classes responsáveis pelo *input/output*;
- *com.palmeiro.ai.jlups.rules*: Contém a lógica associada ao ciclo de vida das regras;
- *com.palmeiro.ai.jlups.config*: Contém classes de configuração do JLups.

7.3.1 *Package* Base

O núcleo deste componente encontra-se implementado no *package* base do componente. Este é composto por dez classes e duas interfaces:

- *Interface JLups* ;
- *Interface ProveStrategy*;
- *JLupsImpl*;
- *ProveVariables*;

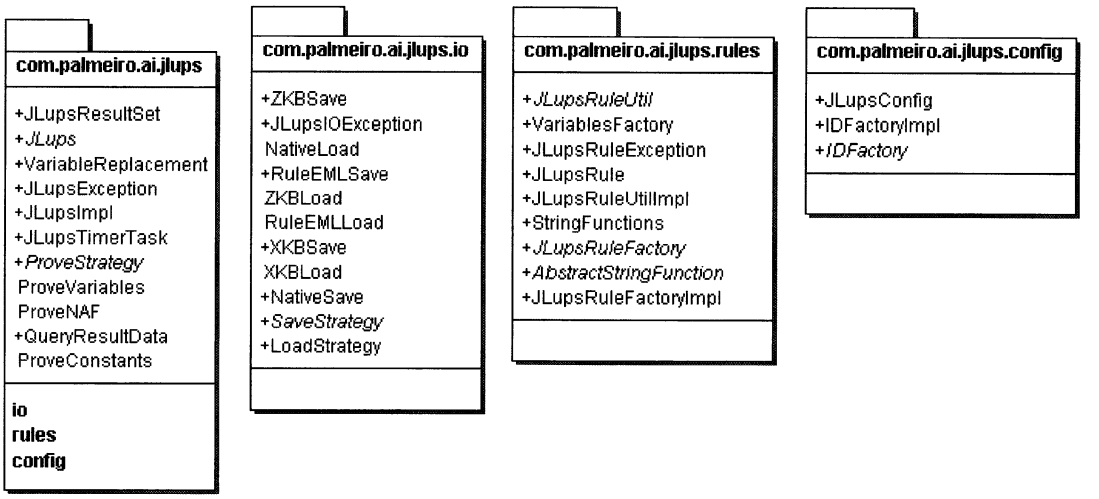

Figura 7.11: JLups - Estrutura de *packages*.

- *ProveConstants*;
- *ProveNAF*;
- *ProveNAF2*;
- *JLupsResultSet*;
- *JLupsTimerTask*;
- *QueryResultData*;
- *VariableReplacement*;
- *JLupsException*.

7.3.1.1 *JLups*

Interface: *com.palmeiro.ai.jlups.JLups*

Descrição: Interface de utilização do componente JLups;

Serviços: Fornece todas as operações descritas na figura 7.1 Casos de Uso. Como funcionalidades acrescidas, pode-se recuperar a configuração do JLups. É fornecida uma implementação base na secção 7.3.1.3;

Protocolo: Não existem constrangimentos na ordem das operações;

Notas: –

Issues: –

Assinatura:

```
public interface JLups {
    /**
     * Executes a JlupsRule, acording to the action specified in the rule.
     * Stands as a strategy for choosing methods invocation.
     * @param rule JLupsRule
     * @throws JLupsException
```

```
 */
public void execute(JLupsRule rule) throws JLupsException;

/**
 * Query in a predetermined state. If state equals QUERY_ALL_STATES a list
 * of JLupsResultSet is returned.
 * @param rule JLupsRule The rule
 * @param state int State
 * @throws JLupsException Exception
 * @return List A List of JLupsResultSet
 */
public List holds(JLupsRule rule, int state) throws JLupsException;

/**
 * Query in present state.
 * @param rule JLupsRule The rule
 * @throws JLupsException Exception
 * @return List A List of JLupsResultSet with zero position filled
 */
public List holds(JLupsRule rule) throws JLupsException;

/**
 * Add an always rule, or a law.
 * @param rule JLupsRule Rule to be added
 * @throws JLupsException
 * @return boolean Return true if added, false if not
 */
public boolean always(JLupsRule rule) throws JLupsException;

/**
 * Assert a rule.
 * @param rule JLupsRule Rule to be asserted
 * @throws JLupsException
 */
public void affirm(JLupsRule rule) throws JLupsException;

/**
 * Removes a law.
 * @param rule JLupsRule Rule to be removed
 * @throws JLupsException
 * @return boolean Returns true rule to be removed is stored correctly
 */
public boolean cancel(JLupsRule rule) throws JLupsException;

/**
 * Removes a rule.
 * @param rule JLupsRule Rule to be removed
 * @throws JLupsException
 */
public void retract(JLupsRule rule) throws JLupsException;

/**
 * Clears the JLups knowledge base.
 * @throws JLupsException
 */
public void clear() throws JLupsException;

/**
 * Loads a JLups knowledge base stored in a file.
 * @param file String file
 * @throws JLupsException
 */
```

```
    public void load(String file) throws JLupsException;

    /**
     * Saves the current knowledge base to a file
     * @param type int Knowledge base store type
     * @param path String
     * @throws JLupsException
     */
    public void save(int type, String path) throws JLupsException;

    /**
     * Updated the knowledge base, advancing in time.
     * @throws JLupsException
     */
    public void update() throws JLupsException;

    /**
     * Adds a timertask to the present knowledge base.
     * @param timer TimerTask
     */
    public void setTimer(TimerTask timer);

    /**
     * Enables/disables the associated timer.
     * @param timer boolean true enables timer. false disables timer.
     */
    public void enableTimer(boolean timer);

    /**
     * Return the current configuration of the knowledge base
     * @throws JLupsException
     * @return JLupsConfig
     */
    public JLupsConfig getConfig() throws JLupsException;
}
```

7.3.1.2 *ProveStrategy*

Interface: *com.palmeiro.ai.jlups.ProveStrategy*

Descrição: Interface de utilização do componente JLups;

Serviços: O objectivo desta interface é fornecer uma forma de provar diversos tipos de condições sem necessidade de se conhecerem os seus detalhes. Foi escolhido o padrão de desenho *Strategy* para implementar esta solução;

Protocolo: Não existem constrangimentos na ordem das operações;

Notas: –

Issues: –

Assinatura:

```
public interface ProveStrategy {
    public boolean prove(List conditions, KnowledgeBase kb)
        throws JLupsException;
    public boolean isProved();
    public List getSubstitutions();
}
```

7.3.1.3 *JLupsImpl*

Componente: *com.palmeiro.ai.jlups.JLupsImpl*

Responsabilidades: Esta classe implementa todas as funcionalidades da interface descrita na secção 7.3.1.1;

Colaboradores: –

Notas: Negação por defeito: Pode estar no corpo ou na cabeça das regras. Um "*assert not bel(X, b)*" não indica que os X não acreditam em b, mas sim que não há X que acreditem em b. Serve para retirar das inferências todos os "*bel(X,b)*", na solução LUPS serve também para indicar onde parar uma prova. A solução escolhida para este problema foi traduzir o significado do estado anterior para o presente, i.e. adicionar à base de conhecimento para o tempo actual todas as regras existentes no estado anterior que sejam coerentes com o presente estado. Existe ainda uma lista de eventos que contém as regras que não irão ser inseridas nos próximos estados;

Issues em Aberto:

1. Negação explicita: Negação de um predicado. Não é uma operação suportada pelo Mandarax. Possíveis soluções poderiam passar pela introdução de predicados semânticos;

2. Variáveis *grounded*: Existem dois tipos de soluções: Prever esta situação no JLups ou introduzi-la no Mandarax. A segunda está fora de contexto para este documento. A primeira poderia passar por detectar estas situações e, de algum modo, obrigar a uma tipificação das variáveis. No exemplo acima, seria necessário definir previamente que o X é um agente e que se pode obter a lista de todos os agentes conhecidos do sistema através do predicado "*agent(X)*". Deste modo, já seria possível obter os agentes para os quais "*bel(X,b)*" é falso. Poderá ser interessante prever uma extensão do tipo: "*not(bel(X:agent, b))*".

7.3.1.4 *ProveVariables*

Componente: *com.palmeiro.ai.jlups.ProveVariables*

Responsabilidades: Esta classe implementa a estratégia de prova com predicados que contêm variáveis;

Colaboradores: –

Notas: –

Issues: –

7.3.1.5 *ProveConstants*

Componente: *com.palmeiro.ai.jlups.ProveVariables*

Responsabilidades: Esta classe implementa a estratégia de prova com predicados que contêm apenas constantes;

Colaboradores: –

Notas: –

Issues: –

7.3.1.6 *ProveNAF*

Componente: *com.palmeiro.ai.jlups.ProveNAF*

Responsabilidades: Esta classe implementa a estratégia de prova para predicados que contêm variáveis com *Negation As Failure* (NAF). Soluciona o problema do *floundering* pois todas as variáveis livres em predicados NAF são tratadas;

Colaboradores: –

Notas: –

Issues: Esta solução apresenta "tempos de prova" altos associados ao crescimento de uma base de conhecimento. O facto de ser uma solução completa, obriga à realização de muitas provas a variáveis livres encontradas. Para mais detalhes consultar o Capítulo 9 Testes, secção 9.1.

7.3.1.7 *ProveNAF2*

Componente: *com.palmeiro.ai.jlups.ProveNAF2*

Responsabilidades: Esta classe implementa a estratégia de prova para predicados que contêm variáveis com NAF. Ao contrário da anterior não soluciona o problema do *floundering* e usa um motor de inferência do Mandarax alterado para lidar com este tipo de prova. Fica a cargo do programador tratar o problema das variáveis livres. Os problemas de desempenho encontrados na classe anterior são eliminados;

Colaboradores: –

Notas: –

Issues: –

7.3.1.8 *VariableReplacement*

Componente: *com.palmeiro.ai.jlups.VariableReplacement*

Responsabilidades: Classe auxiliar para armazenar uma substituição de uma variável por um valor;

Colaboradores: ProveNAF;

Notas: –

Issues: –

7.3.1.9 *QueryResultData*

Componente: *com.palmeiro.ai.jlups.QueryResultData*

Responsabilidades: Classe auxiliar para armazenar a gerir um conjunto de variáveis/valores.

Colaboradores: ProveNAF;

Notas: –

Issues: –

7.3.1.10 *JLupsResultSet*

Componente: *com.palmeiro.ai.jlups.JLupsResultSet*

Responsabilidades: Esta classe implementa a interface *org.mandarax.kernel.ResultSet.* Fornece um mecanismo *standard* de tratamento do resultado de uma *query*;

Colaboradores: JLupsRule e VariablesFactory;

Notas: O método com a assinatura "*public Object getResult(VariableTerm term) throws InferenceException*" não se encontra implementado;

Issues: –

7.3.1.11 *JLupsTimerTask*

Componente: *com.palmeiro.ai.jlups.JLupsTimerTask*

Responsabilidades: Esta classe extende a classe *java.util.TimerTask* e apresenta o objectivo de invocar o método *update* de um objecto *JLups*, num determinado intervalo de tempo, de forma a iniciar um novo estado e garantir a evolução do sistema mesmo sem estímulos externos;

Colaboradores: JLups;

Notas: A presente classe é fornecida como exemplo de uma possível utilização desta funcionalidade, necessitando de ser explicitamente associada ao JLups para que funcione;

Issues: –

7.3.1.12 *JLupsException*

Componente: *com.palmeiro.ai.jlups.JLupsException*

Responsabilidades: Esta classe representa uma excepção no motor do componente JLups;

Colaboradores: Exception;

Notas: –

Issues: –

7.3.2 *Package* de Regras

O presente *package* contém duas interfaces e sete classes:

- *Interface JLupsRuleUtil* ;
- *Interface JLupsRuleFactory*;
- *JLupsRuleUtilImpl*;
- *JLupsRuleFactoryImpl*;
- *JLupsRule*;

- *JLupsRuleException*;
- *AbstractStringFunction*;
- *StringFunctions*;
- *VariablesFactory*;

7.3.2.1 *JLupsRuleFactory*

Interface: *com.palmeiro.ai.jlups.rules.JLupsRuleFactory*

Descrição: Interface de utilização para construção de regras JLups. Esta interface implementa o padrão de desenho *Factory Method* [Gra98, pp. 89-98], juntamente com a respectiva implementação descrita na secção 7.3.2.5;

Serviços: Fornece a assinatura para as diferentes implementações para construção de regras. O *input* deste método é uma regra LUPS em formato textual;

Protocolo: Não existem constrangimentos na ordem das operações;

Notas: –

Issues: Esta fábrica apresenta ainda um comportamento limitado ao nível gramatical, trabalhando apenas com regras simples.

Assinatura:

```
public interface JLupsRuleFactory {
     public JLupsRule createRule(String rule)throws JLupsRuleException;
}
```

7.3.2.2 *JLupsRuleUtil*

Interface: *com.palmeiro.ai.jlups.rules.JLupsRuleUtil*

Descrição: Interface que fornece um conjunto de funcionalidades sobre regras. É fornecida uma implementação descrita na secção 7.3.2.4;

Serviços: Realiza substituições de variáveis; Transforma factos em *queries*; Nega regras; Verifica negações e recupera o corpo de uma regra;

Protocolo: Não existem constrangimentos na ordem das operações;

Notas: –

Issues: –

Assinatura:

```
public interface JLupsRuleUtil {
    public ClauseSet performSubstitution(
        JLupsRule rule,
        String variable,
        String value) throws JLupsException ;
    public Fact createQueryFact(Fact fact) throws JLupsException ;
    public JLupsRule negateRule(JLupsRule rule) throws JLupsRuleException;
    public boolean isNegated(JLupsRule rule) throws JLupsRuleException;
    public List getBodyToProve(JLupsRule rule) throws JLupsRuleException;
}
```

7.3.2.3 *JLupsRule*

Componente: *com.palmeiro.ai.jlups.rules.JLupsRule*

Responsabilidades: Esta classe representa uma regra simples JLups. Encapsula um objecto *org.mandarax.kernel.ClauseSet* que representa uma regra Mandarax ao mais alto nível;

Colaboradores: –

Notas: –

Issues: –

7.3.2.4 *JLupsRuleUtilImpl*

Componente: *com.palmeiro.ai.jlups.rules.JLupsRuleUtilImpl*

Responsabilidades: Implementação da interface *JLupsRuleUtil*;

Colaboradores: –

Notas: –

Issues: –

7.3.2.5 *JLupsRuleFactoryImpl*

Componente: *com.palmeiro.ai.jlups.rules.JLupsRuleFactoryImpl*

Responsabilidades: Fábrica de regras JLups com base em regra LUPS;

Colaboradores: –

Notas: A presente classe realiza um *parsing* a uma regra LUPS e constrói uma regra Mandarax, encapsulando-a numa regra JLups (*wrapper*). O processo de *parsing* está na presente versão limitado ao nível da complexidade das regras;

Issues: –

7.3.2.6 *VariablesFactory*

Componente: *com.palmeiro.ai.jlups.rules.VariablesFactory*

Responsabilidades: Esta classe é responsável pela criação de variáveis. É usada para substituição de variáveis, em que cada uma apresente a forma de um caracter alfanumérico. Implementa ainda o padrão *singleton*;

Colaboradores: –

Notas: –

Issues: –

7.3.2.7 *JLupsRuleException*

Componente: *com.palmeiro.ai.jlups.rules.JLupsRuleException*

Responsabilidades: Esta classe representa uma extensão a uma excepção base, indicando uma excepção ao nível de regras JLups;

Colaboradores: JLupsException;

Notas: –

Issues: –

7.3.2.8 *AbstractStringFunction*

Componente: *com.palmeiro.ai.jlups.rules.JLupsRuleException*

Responsabilidades: Classe abstracta de funções Mandarax extendendo a classe *AbstractFunction* desta biblioteca;

Colaboradores:

Notas: –

Issues: –

7.3.2.9 *StringFunctions*

Componente: *com.palmeiro.ai.jlups.rules.StringFunctions*

Responsabilidades: Esta classe extende a classe acima descrita e implementa uma função Mandarax para concatenações de Strings. Esta função pode ser usada em regras para concatenar várias variáveis;

Colaboradores: AbstractStringFunction;

Notas: –

Issues: –

7.3.3 *Package* de IO

O *package* de IO contém uma interface e dez classes:

- *Interface SaveStrategy* ;
- *LoadStrategy*;
- *NativeSave*;
- *RuleEMLSave*;
- *ZKBSave*;
- *XKBSave*;
- *NativeLoad*;
- *RuleEMLLoad*;

- *ZKBLoad*;
- *XKBLoad*;
- *JLupsIOException*;

7.3.3.1 *SaveStrategy*

Interface: *com.palmeiro.ai.jlups.io.SaveStrategy*

Descrição: Classe abstracta para gravação de bases de conhecimento JLups, implementando o padrão de desenho *Strategy* [Gra98, p. 371-377]. As respectivas implementações encontram-se descritas nas secções 7.3.3.3, 7.3.3.4, 7.3.3.5 e 7.3.3.6;

Serviços: Fornece a assinatura para a gravação de bases de conhecimento JLups;

Protocolo: Não existem constrangimentos na ordem das operações;

Notas: –

Issues: –

Assinatura:

```
public abstract class SaveStrategy {
    public abstract void save(String path) throws JLupsIOException;
}
```

7.3.3.2 *LoadStrategy*

Componente: *com.palmeiro.ai.jlups.io.LoadStrategy*

Responsabilidades: Classe responsável pela leitura de bases de conhecimento JLups, implementando o padrão de desenho *Strategy*. As respectivas implementações encontram-se descritas nas secções 7.3.3.7, 7.3.3.8, 7.3.3.9 e 7.3.3.10;

Colaboradores: NativeLoad;

Notas: –

Issues: –

7.3.3.3 *NativeSave*

Componente: *com.palmeiro.ai.jlups.io.NativeSave*

Responsabilidades: Esta classe faz parte da implementação do padrão de desenho *Strategy* cuja interface está definida na classe *SaveStrategy*. A estratégia desta implementação é dedicada à gravação de ficheiros nativos LUPS;

Colaboradores: SaveStrategy;

Notas: –

Issues: –

7.3.3.4 *RuleEMLSave*

Componente: *com.palmeiro.ai.jlups.io.RuleEMLSave*

Responsabilidades: Esta classe faz parte da implementação do padrão de desenho *Strategy* cuja interface está definida na classe *SaveStrategy*. A estratégia desta implementação é dedicada à gravação de ficheiros do tipo RULE_EML;

Colaboradores: SaveStrategy

Notas: –

Issues: –

7.3.3.5 *ZKBSave*

Componente: *com.palmeiro.ai.jlups.io.ZKBSave*

Responsabilidades: Esta classe faz parte da implementação do padrão de desenho *Strategy* cuja interface está definida na classe *SaveStrategy*. A estratégia desta implementação é dedicada à gravação de ficheiros do tipo ZKB;

Colaboradores: SaveStrategy;

Notas: –

Issues: –

7.3.3.6 *XKBSave*

Componente: *com.palmeiro.ai.jlups.io.XKBSave*

Responsabilidades: Esta classe faz parte da implementação do padrão de desenho *trategy* cuja interface está definida na classe *SaveStrategy*. A estratégia desta implementação é dedicada à gravação de ficheiros do tipo XKB;

Colaboradores: SaveStrategy;

Notas: –

Issues: –

7.3.3.7 *NativeLoad*

Componente: *com.palmeiro.ai.jlups.io.NativeLoad*

Responsabilidades: Esta classe faz parte da implementação do padrão de desenho *Strategy* cuja interface está definida na classe *LoadStrategy*. A estratégia desta implementação é dedicada à leitura de ficheiros nativos LUPS;

Colaboradores: LoadStrategy;

Notas: –

Issues: –

7.3.3.8 *RuleEMLLoad*

Componente: *com.palmeiro.ai.jlups.io.RuleEMLLoad*

Responsabilidades: Esta classe faz parte da implementação do padrão de desenho *Strategy* cuja interface é definida na classe *LoadStrategy*. A estratégia desta implementação é dedicada à leitura de ficheiros gravados com o tipo RULE_EML;

Colaboradores: LoadStrategy;

Notas: –

Issues: –

7.3.3.9 *ZKBLoad*

Componente: *com.palmeiro.ai.jlups.io.ZKBLoad*

Responsabilidades: Esta classe faz parte da implementação do padrão de desenho *Strategy* cuja interface é definida na classe *LoadStrategy*. A estratégia desta implementação é dedicada à leitura de ficheiros gravados com o tipo ZKB;

Colaboradores: LoadStrategy;

Notas: –

Issues: –

7.3.3.10 *XKBLoad*

Componente: *com.palmeiro.ai.jlups.io.XKBLoad*

Responsabilidades: Esta classe faz parte da implementação do padrão de desenho *Strategy* cuja interface é definida na classe *LoadStrategy*. A estratégia desta implementação é dedicada à leitura de ficheiros gravados com o tipo XKB;

Colaboradores: LoadStrategy;

Notas: –

Issues: –

7.3.3.11 *JLupsIOException*

Componente: *com.palmeiro.ai.jlups.JLupsIOException*

Responsabilidades: Esta classe representa uma extensão a uma excepção base, indicando uma excepção ao nível de *input/output* JLups, nomeadamente na leitura e gravação da base de conhecimento;

Colaboradores: JLupsException;

Notas: –

Issues: –

7.3.4 *Package* de Configurações

O *package* de configurações contém uma interface e duas classes:

- *Interface IDFactory* ;
- *JLupsConfig*;
- *IDFactoryImpl*;

7.3.4.1 *IDFactory*

Interface: *com.palmeiro.ai.jlups.config.IDFactory*

Descrição: Interface para geração de identificadores de bases de conhecimento JLups;

Serviços: Fornece a assinatura para a geração de identificadores;

Protocolo: –

Notas: –

Issues: –

Assinatura:

```
public interface IDFactory {
    public String generateID();
}
```

7.3.4.2 *JLupsConfig*

Componente: *com.palmeiro.ai.jlups.config.JLupsConfig*

Responsabilidades: Esta classe representa uma configuração residente no ficheiro principal de uma base de conhecimento JLups;

Colaboradores: IDFactoryImpl;

Notas: –

Issues: –

7.3.4.3 *IDFactoryImpl*

Componente: *com.palmeiro.ai.jlups.config.IDFactoryImpl*

Responsabilidades: Implementação base de um gerador de identificadores JLups;

Colaboradores: IDFactory;

Notas: –

Issues: –

7.3.5 Cenários

7.3.5.1 Adição de uma nova Regra

Operação: *boolean affirm(JLupsRule rule) throws JLupsException*

Descrição: Adiciona uma nova regra JLups

Actores: Cliente do componente

Input: (*JLupsRule*) Uma regra JLups

Output: (*boolean*) Retorna verdadeiro se a regra for inserida, ou falso caso contrário

Excepções: (*JLupsException*) É lançada uma excepção em caso de erro

Condições Iniciais: –

Condições Finais: Se as condições forem verdadeiras, adiciona-se uma nova regra à base de conhecimento. Se for um evento, esta regra só permanecerá por um estado.

Modelo de Interacção de Componentes:

As figuras 7.12 e 7.13 descrevem os componentes e respectivas interacções.

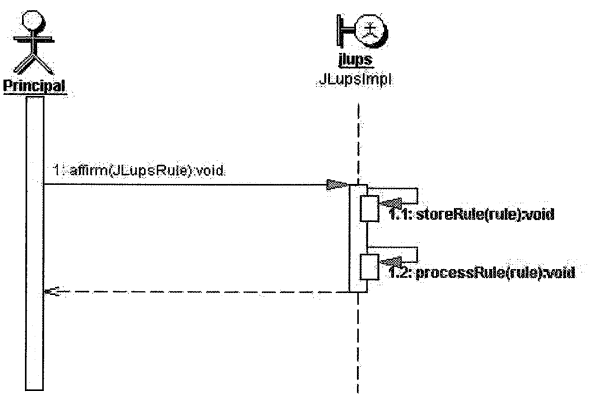

Figura 7.12: Assert - Diagrama de Sequência.

7.3.5.2 Remoção de uma Regra

Operação: *boolean retract(JLupsRule rule) throws JLupsException*

Descrição: Remove uma regra JLups

Actores: Cliente do componente

Input: (*JLupsRule*) Uma regra JLups

Output: (*boolean*) Se a regra for removida devolve verdadeira, caso contrário devolve falso

Excepções: (*JLupsException*) É lançada uma excepção em caso de erro

Condições Iniciais: –

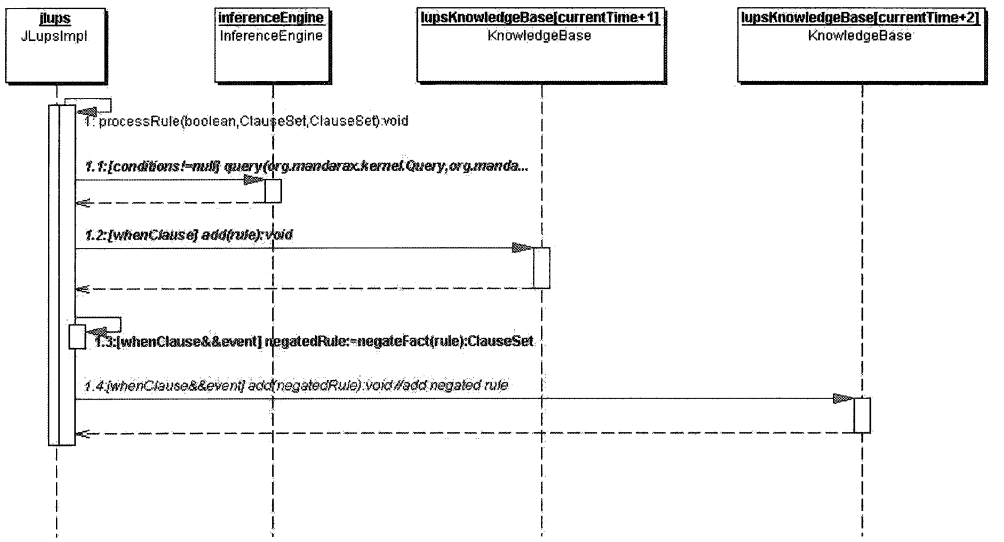

Figura 7.13: Processamento de uma regra - Diagrama de Sequência.

Condições Finais: Se as condições forem verdadeiras, insere-se a regra negada por defeito à base de conhecimento. Se for um evento e esta regra existir, insere-se novamente no estado *t+2*.

Modelo de Interacção de Componentes:

A figura 7.14 descreve os componentes e respectivas interacções.

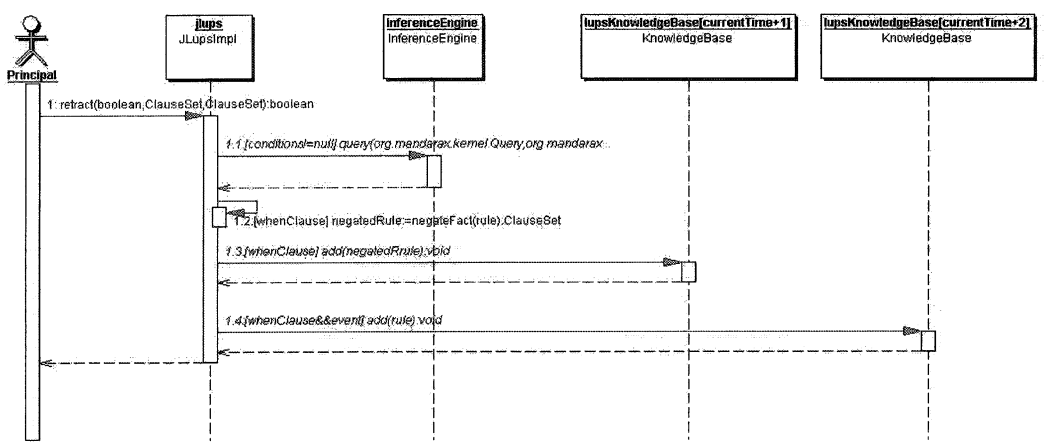

Figura 7.14: Retract - Diagrama de Sequência.

7.3.5.3 Introdução de uma Lei

Operação: *boolean always(JLupsRule rule) throws JLupsException*

Descrição: Introduz uma regra permanente

Actores: Cliente do componente

Input: (*JLupsRule*) Uma regra JLups

Output: (*boolean*) Se a regra for armazenada devolve verdadeiro[3], caso contrário devolve falso.

Excepções: (*JLupsException*) É lançada uma excepção em caso de erro

Condições Iniciais: –

Condições Finais: Armazena-se a regra numa lista de leis. Doravante e por cada *update*, se as condições forem verdadeiras adiciona-se a regra a cada novo estado.

Modelo de Interacção de Componentes:

A figura 7.15 descreve os componentes e respectivas interacções.

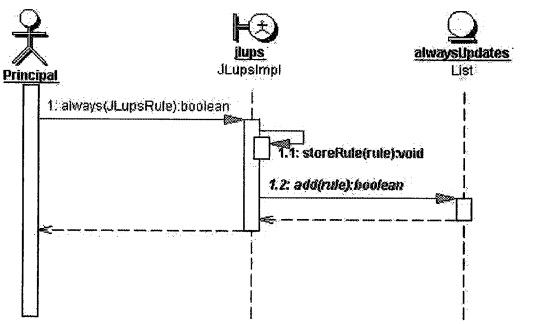

Figura 7.15: Always - Diagrama de Sequência.

7.3.5.4 Cancelamento de uma Lei

Operação: *boolean cancel(JLupsRule rule) throws JLupsException*

Descrição: Cancela uma regra permanente

Actores: Cliente do componente

Input: (*JLupsRule*) Uma regra JLups

Output: (*boolean*) Se a regra for armazenada na lista de cancelamento devolve verdadeiro, caso contrário devolve falso. Análogo ao processo de introdução de leis.

Excepções: (*JLupsException*) É lançada uma excepção em caso de erro

Condições Iniciais: –

Condições Finais: Armazena-se a regra numa lista de cancelamentos. Doravante, para todos os novos estados, realiza-se uma filtragem à lista de leis com a presente lista.

Modelo de Interacção de Componentes:

A figura 7.16 descreve os componentes e respectivas interacções.

[3] *"As per the general contract of the Collection.add method"*, como descrito nos javadocs da classe *java.util.List*.

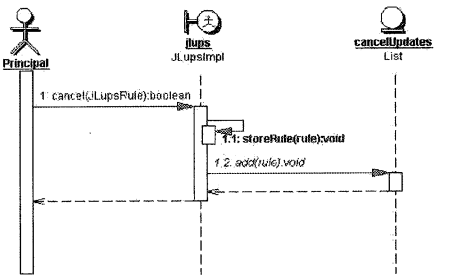

Figura 7.16: Cancel - Diagrama de Sequência.

7.3.5.5 Actualização

Operação: *boolean update() throws JLupsException*

Descrição: Processo de actualização da base de conhecimento JLups. Incrementa-se o estado, criando uma nova base de conhecimento Mandarax. Calcula-se a diferença entre a lista de leis e a lista de cancelamentos, e processam-se as regras que não foram canceladas.

Actores: Cliente do componente

Input: –

Output: (*void*)

Excepções: (*JLupsException*) É lançada uma excepção em caso de erro

Condições Iniciais: –

Condições Finais: Um novo estado JLups com as leis válidas inseridas

Modelo de Interacção de Componentes:

A figura 7.17 descreve os componentes e respectivas interacções.

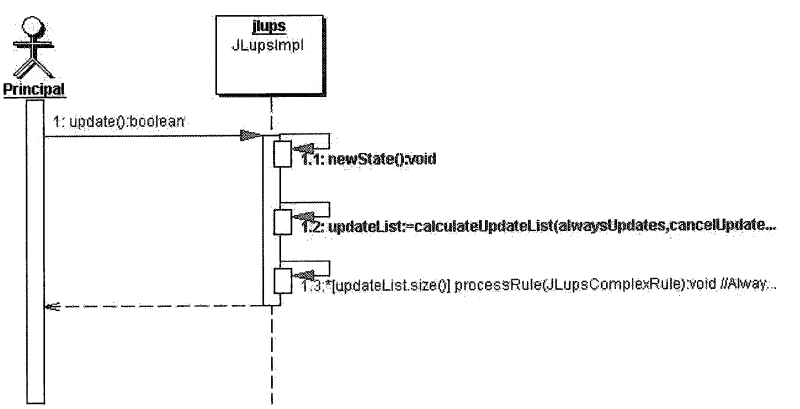

Figura 7.17: Update - Diagrama de Sequência.

7.3.5.6 *Query* ao JLups

Operação: *ResultSet holds(JLupsRule rule, int state) throws JLupsException*

Descrição: Consulta ao JLups

Actores: Cliente do componente

Input: (*JLupsRule*) Uma regra JLups e o estado desejado[4]

Output: (*ResultSet*) Um *ResultSet* com o resultado da *query* ou uma lista de objectos *ResultSet*, um índice por cada estado

Excepções: (*JLupsException*) É lançada uma excepção em caso de erro

Condições Iniciais: –

Condições Finais: –

Modelo de Interacção de Componentes:

As figuras 7.18 e 7.19 descrevem os componentes e respectivas interacções.

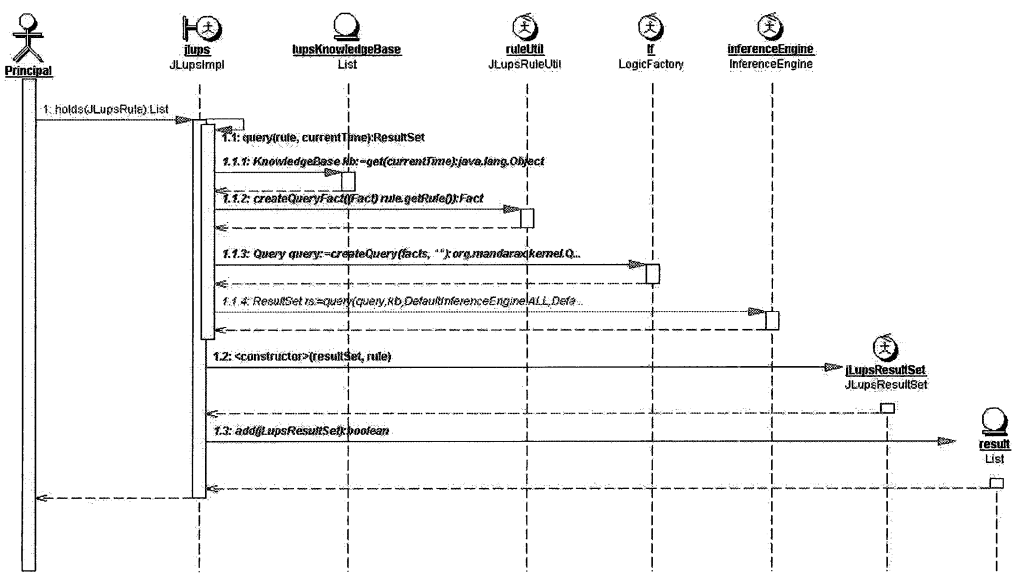

Figura 7.18: Holds sem Estado - Diagrama de Sequência.

7.3.5.7 Gravação da Base de Conhecimento

Operação: *void save(int type, String path) throws JLupsException*

Descrição: Gravação da base de conhecimento JLups. A gravação consiste na geração de um conjunto de ficheiros, nomeadamente um ficheiro principal (descrito abaixo), um ficheiro correspondente do tipo NATIVE, e um ficheiro por cada estado de acordo com o tipo definido - se diferente de NATIVE.

Actores: Cliente do componente

[4]Opcional.

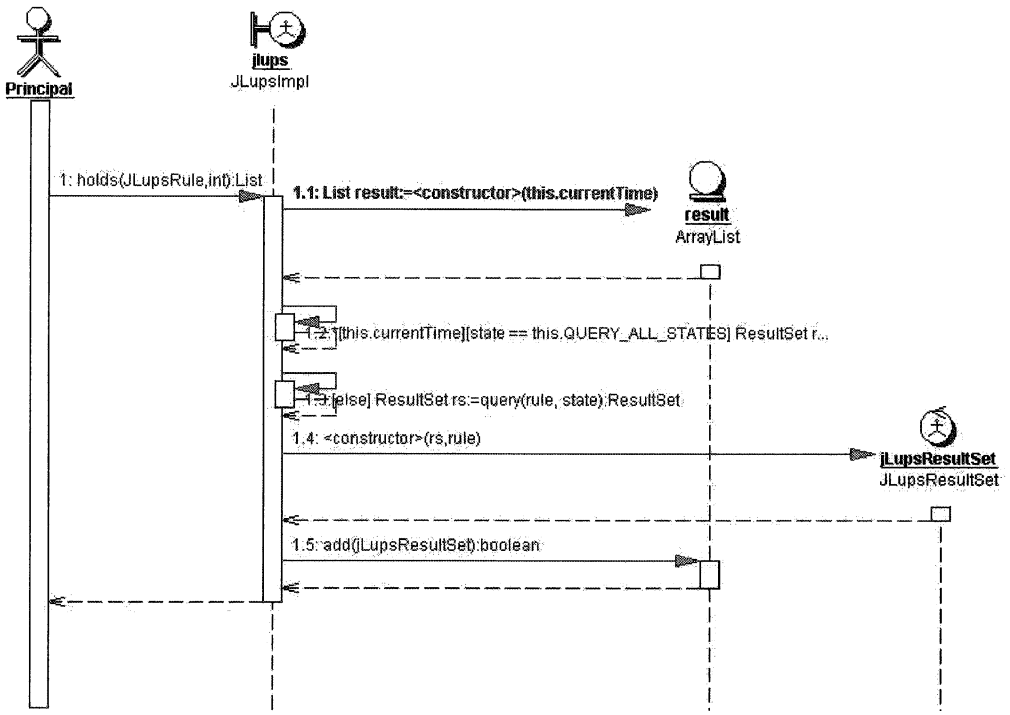

Figura 7.19: Holds com Estado - Diagrama de Sequência.

Input: O tipo e um título para a base de conhecimento. O tipo pode variar entre NATIVE, RULE_EML, XKB e ZKB

Output: (*void*)

Excepções: (*JLupsException*) É lançada uma excepção em caso de erro

Condições Iniciais: –

Condições Finais: Uma base de conhecimento gravada para memória persistente

Modelo de Interacção de Componentes:

O ficheiro principal associado a cada gravação de uma base de conhecimento JLups apresenta o seguinte formato:

```
<jlups_kb>
<!-- [RULE_EML|XKB|ZKB|NATIVE] -->
<type></type>
<date>
<creation></creation>
<updated></updated>
</date>
<!-- kb id -->
<id></id>
<!-- Path to LUPS compatible kb -->
<native></native>
<number_of_states></number_of_states>
<always_list>
<!-- a text representation -->
<entry></entry>
...
```

```
</always_list>
<cancel_list>
<entry></entry>
...
</cancel_list>
</jlups_kb>
```

A figura 7.21 apresenta o *schema* XML de uma base de conhecimento JLups, enquanto a figura 7.20 descreve os componentes e respectivas interacções.

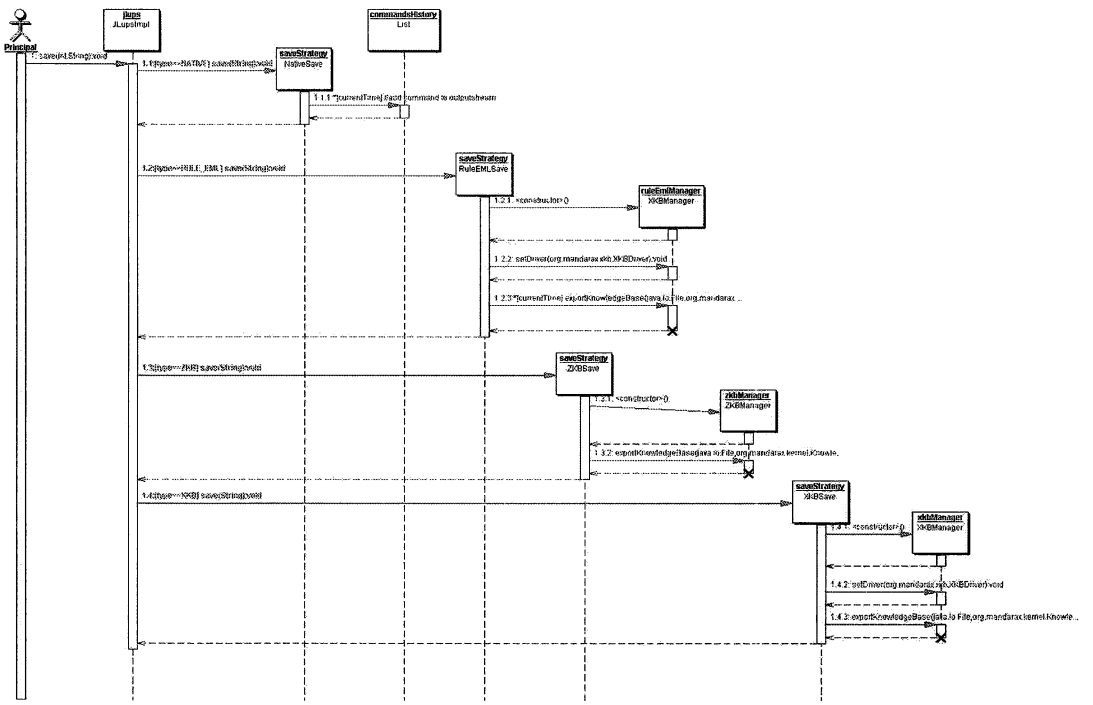

Figura 7.20: Save - Diagrama de Sequência.

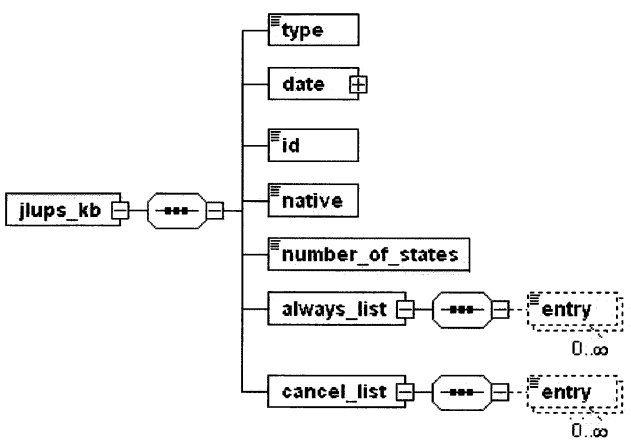

Figura 7.21: Estrutura de um ficheiro principal de uma base de conhecimento.

Caso o tipo de gravação escolhida tenha sido nativa, é também gerado um outro ficheiro com formato LUPS, Neste tipo de ficheiro as linhas que começam com o símbolo de

percentagem são comentários, e as restantes são os comandos introduzidos (ver exemplo seguinte).

```
% Example
% Update 1
always (not jail(X) <- abn(X)) when (not repC,not repP).
newUpdate.

% Update 2
assert not repC.
newUpdate.
```

O conteúdo deste ficheiro resume-se a uma listagem dos comandos introduzidos.

7.3.5.8 Leitura de uma Base de Conhecimento

7.3.5.9 Descrição do Cenário

Operação: *void load(String file) throws JLupsException*

Descrição: Carregamento de uma base de conhecimento JLups. A leitura consiste na interpretação de um ficheiro de entrada. Este ficheiro pode apresentar o formato descrito na secção anterior, ou ser um ficheiro original LUPS[5]. Como primeiro passo é interpretado o ficheiro principal no método *loadKBConfig*, construindo um objecto *JLupsConfig*. A partir deste pode obter-se todos os detalhes da base de conhecimento e carregar cada estado para memória. Para esta tarefa são usadas estratégias que escondem as várias implementações para os vários tipos diferentes de formato.

Actores: Cliente do componente

Input: Um caminho para um ficheiro

Output: (*void*)

Excepções: (*JLupsException*) É lançada uma excepção em caso de erro

Condições Iniciais: –

Condições Finais: Uma base de conhecimento carregada para memória

Modelo de Interacção de Componentes

A figura 7.22 descreve os componentes e respectivas interacções.

7.3.6 Mecanismos

Tratamento de Excepções

Todos os erros ou situações anómalas são tratadas por excepções. Estas excepções, descritas em 7.3.1.12, 7.3.2.7 e 7.3.3.11 contêm uma mensagem indicadora da situação de erro.

[5]Existem um elevado nível de redundância nesta situação, pois é possível ler a mesma base de conhecimento de duas formas, através do ficheiro nativo LUPS ou do ficheiro XML criado pelo JLups, sendo que este último também contém o ficheiro nativo. Esta ausência de determinismo justifica-se por questões de compatibilidade com os ficheiros LUPS, sendo resolvida através de uma *Strategy*.

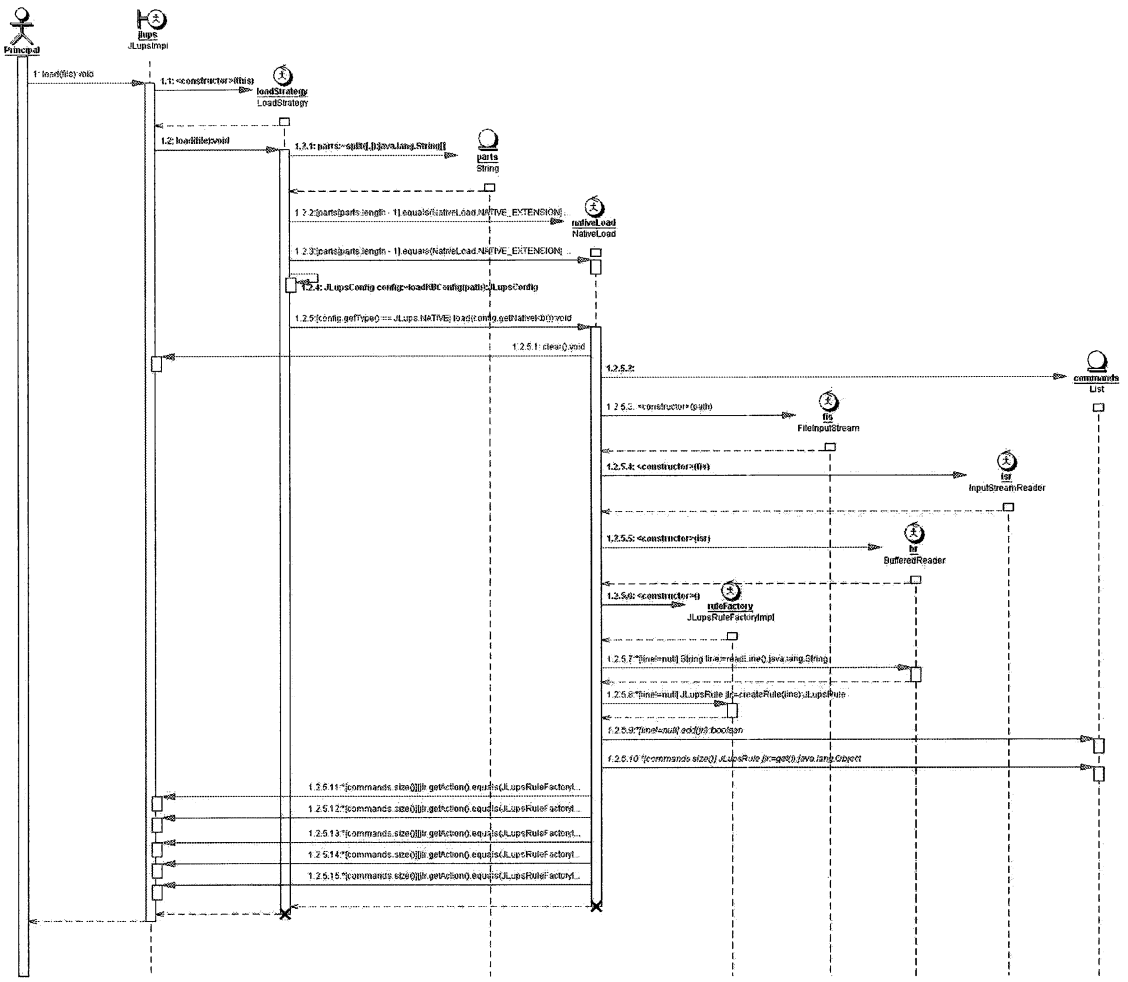

Figura 7.22: Load - Diagrama de Sequência.

Logging

O mecanismo de *logging* é realizado através da classe *java.util.logging.Logger*. Existe ainda uma parametrização do nível de detalhe, que ajuda não só o nível de *debugging* mas também a avaliação de históricos e análise de resultados.

7.4 Vista de Execução

7.4.1 Processos

7.4.1.1 Actualizações Dinâmicas

O processo de actualizações dinâmicas é fornecido através de duas operações - do componente JLups. A primeira introduz um *timer* ao JLups enquanto a segunda fornece forma de ligar ou desligar o referido processo.

Operação1: *void setTimer(TimerTask timer)*

Descrição: Adiciona um *timer* ao JLups. Apresenta a mesma funcionalidade de segundo construtor, tendo o objectivo de poder automatizar operações, analogamente ao objecto *JLupsTimerTask* que em determinado espaço de tempo realiza

uma operação de *update*. Através desta operação é possível construir um *timer* por medida, que automatize um determinado conjunto de operações definidas sobre o JLups.

Actores: Cliente do componente

Input: Um classe construída por medida que extenda a classe *java.util.TimerTask*

Output: (*void*)

Excepções: –

Condições Iniciais: –

Condições Finais: –

Operação2: *void enableTimer(boolean timer)*

Descrição: Permite ligar ou desligar o *timer*. Se este atributo não tiver definido o método irá apresentar qualquer resultado.

Actores: Cliente do componente

Input: Uma variável booleana que indica se o *timer* deve iniciar-se (*true*) ou deve desligar-se (*false*)

Output: (*void*)

Excepções: –

Condições Iniciais: –

Condições Finais: –

O exemplo do *timer* fornecido nesta implementação, a classe *JLupsTimerTask*, está descrita abaixo no diagrama de sequência da figura 7.23.

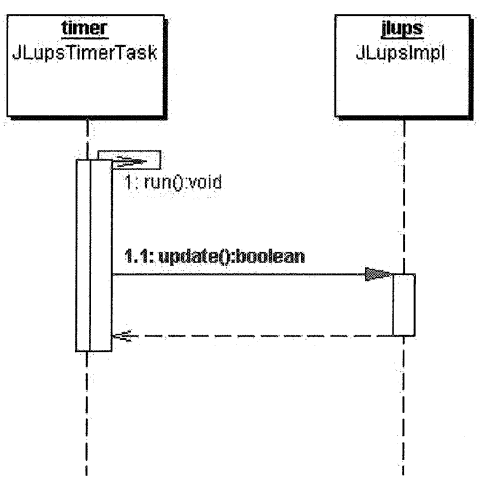

Figura 7.23: Processo JLupsTimerTask - Diagrama de Sequência.

7.5 Vista de Instalação

7.5.1 Estrutura de Componentes

O componente JLups é distribuído no formato *standard* para aplicações *desktop*, encapsulado num ficheiro *jlups.jar*.

Este componente foi desenvolvido com plataforma Java SE 5, apresentando como requisito mínimo de uso a versão referida da linguagem Java e uma dependência.

A única dependência é para o projecto Mandarax[6], que é uma biblioteca *open source* para dedução de regras, fornecendo uma estrutura para definir, gerir e realizar inferências sobre bases de conhecimento. É ainda uma implementação Java pura de um *rule engine* que suporta múltiplos tipos de factos e regras baseadas em reflexão, bases de dados, EJB, entre outros. Também fornece um motor de inferência Java EE[7] *compliant* usando *backward chaining*. A versão usada nesta *release* é a 3.4.

O JLups encontra-se na *release candidate* 9 versão 1.0 (jlups-1.0-rc9).

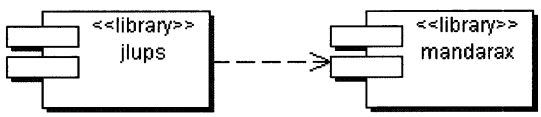

Figura 7.24: JLups - Diagrama de Componentes

7.6 Vista de Implementação

7.6.1 Estrutura de Classes

As classes descritas nas vistas anteriores estão representadas no diagrama de classes das figuras 7.25, 7.26, 7.27 e 7.28. Qualquer desenvolvimento que se efectue posteriormente deve respeitar a lógica de *packages* usados, adicionando novos *packages* se necessário.

7.6.2 Regras de Implementação

Devem ser seguidas as regras definidas no Capítulo 6, secção 6.7.5 Regras de Implementação. Cada classe deve ainda apresentar o cabeçalho seguido de nome do *package*, código de *imports*, descrição e autoria da classe. Um exemplo é descrito abaixo.

```
/**
 * JLups
 *
 * Copyright (C) 2004 José Palmeiro (jpalmeiro@gmail.com)
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
```

[6]Consultar http://mandarax.sourceforge.net/

[7]Consultar http://java.sun.com/j2ee//

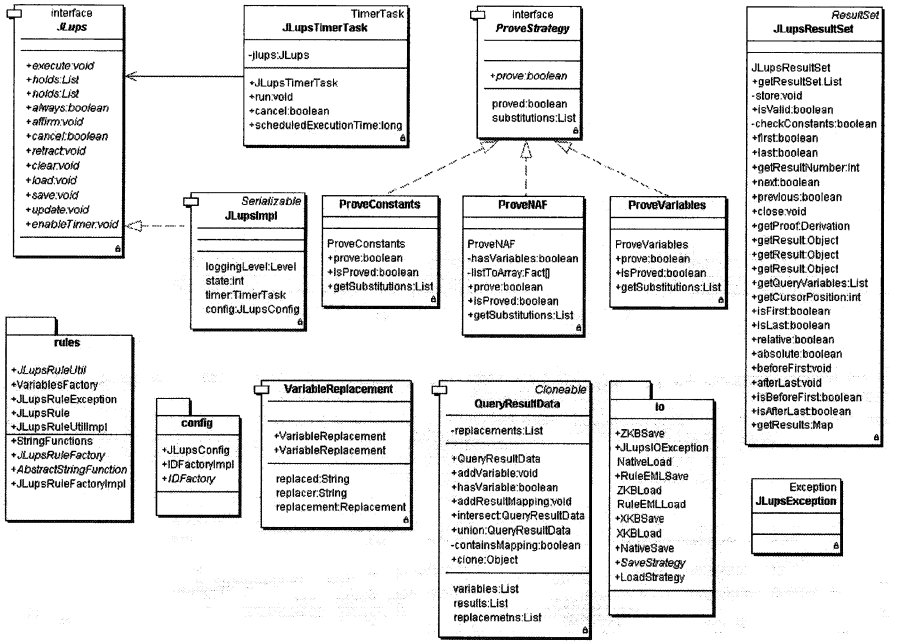

Figura 7.25: *Package com.palmeiro.ai.jlups* - Diagrama de Classes.

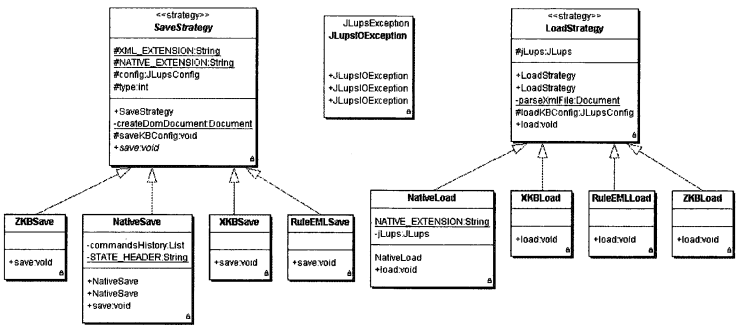

Figura 7.26: *Package com.palmeiro.ai.jlups.io* - Diagrama de Classes.

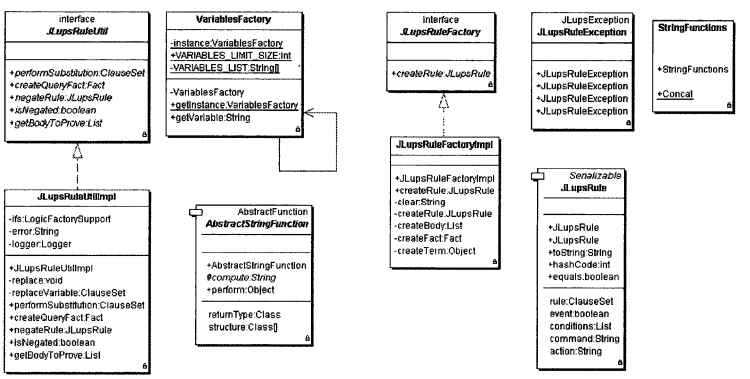

Figura 7.27: *Package com.palmeiro.ai.jlups.rules* - Diagrama de Classes.

```
* You should have received a copy of the GNU General Public License
* along with this program; if not, write to the Free Software
* Foundation, Inc., 59 Temple Place -
* Suite 330, Boston, MA  02111-1307, USA. *
```

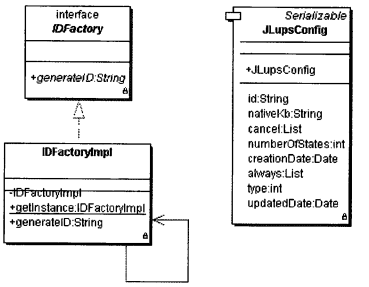

Figura 7.28: *Package com.palmeiro.ai.jlups.config* - Diagrama de Classes.

```
 */

package com.palmeiro.ai.jlups...;

import ...;

/**
 * Breve descrição das funcionalidades e objectivos da classe
 *
 * @author   Copyright      : jpalmeiro
 * @author   Created by     : jpalmeiro
 * @author   Authored by    : $Author: Jose Palmeiro $
 * @version  Revision       : $Revision: 1.1 $ $Date: 2004/11/19 01:17:44 $
*/

...
```

7.7 Resumo

O Capítulo 8, secção 8.4, resume o componente JLups.

Capítulo 8

Resumo Arquitectural

Nos três capítulos anteriores foi apresentado o detalhe de uma arquitectura de agentes - Susy. Explorada num contexto de *Information Retrieval* (IR), esta pesquisa foca-se no uso de tecnologia de agentes em aplicações de âmbito empresarial. A solução propõe o uso do padrão MVC com a camada *model* orientada a agentes. Alguns agentes mantêm bases de conhecimento modeladas em DLP, enquanto outros são parte integrante de uma SOA. Ao longo deste capítulo ir-se-ão resumir os pontos mais importantes da arquitectura proposta, a um nível bastante alto, apresentado a suas características, componentes e fluxos mais significativos.

8.1 Introdução

A solução proposta explora um sistema de IR genérico chamado Susy[1]. Este sistema está disponível sob a forma de uma aplicação *web* com dois tipos de utilizadores, um utilizador normal e um administrador. Cada utilizador registado no sistema dispõe de um agente pessoal que cooperando com outros agentes irá participar na recuperação e refinamento da informação pedida.

A presente *release* contém os seguintes casos de uso:

- Autenticação: Autenticação de um utilizador no sistema;
- Pesquisa: O caso de uso mais significativo com três tipo de pesquisa: Pesquisa Simples; Pesquisa Automatizada com assimilação directa de conhecimento; e Pesquisa Interactiva onde o utilizador selecciona o conhecimento que o seu agente irá adquirir, apresentando um papel pedagógico;
- Sumarização: Sistema de sumarização de documentos;
- Gestão de Crenças: Um utilizador pode adicionar e remover crenças, interagindo directamente com o seu agente;
- Administração[2]: Um administrador pode interagir com o sistema através da alteração das base de conhecimento de cada agente;
- Mensagens: O sistema armazena e permite a consulta de mensagens provenientes de agentes de serviços.

[1]Note-se que o foco desta dissertação é na arquitectura da solução e não na exploração e avaliação de um sistema de IR.

[2]Em fase de testes.

A plataforma Susy, para além de um sistema simples de IR, é um aplicação de *software* complexa e dividida em várias camadas de abstracção, sendo composta por vários estilos arquitecturais (ver figura 6.1):

- *Web-layer*: Uma aplicação *web* implementada com a Java EE[3] fornecendo e controlando o acesso e navegação no sistema;
- *Agents-layer*: Um MAS implementado com a JADE[4], que controla o sistema e as intenções dos agentes, representando a lógica de negócio do sistema;
- *Knowledge Representation-layer*: Uma camada baseada em LUPS[5] que modela o conhecimento e lógica interna de alguns agentes.

As camadas acima referidas estão descritas nas secções 8.2, 8.3 and 8.4, respectivamente. A secção 8.5 apresenta ainda a vista de *deployment* dos diversos componentes que compõem a plataforma.

8.2 Arquitectura Web

A primeira camada é uma clássica aplicação *web* desenvolvida com a Java EE (ver diagrama de *deployment* na figure 8.4), actuando como ponto de entrada no sistema.

Foi usada uma aproximação *Model 2*, exemplificada na figura 6.2 e 8.1. Uma arquitectura *Model 2* introduz a noção de um controlador (através de *servlets*) entre o *browser* e as páginas finais. Este modelo arquitectural é um boa prática para aplicações interactivas e promove a utilização do padrão de desenho MVC (ver secção 6.4.2.1 *Model View Controller*), para separar as diferentes camadas do sistema. Como o nome indica, este padrão considera três camadas.

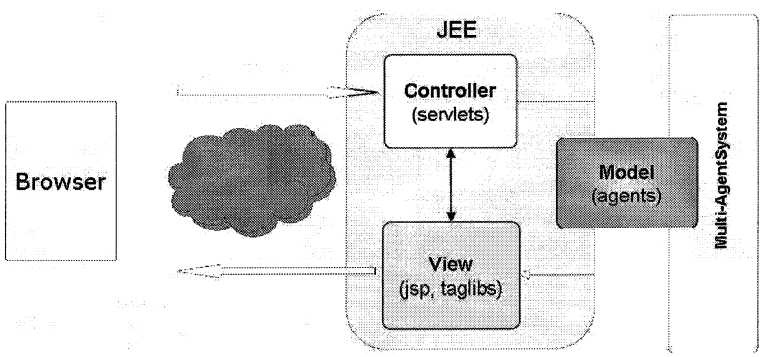

Figura 8.1: Plataforma Susy - MVC e MAS.

O controlador centraliza a lógica de recepção de pedidos e de entrega de respostas baseado no URL de pedido, nos parâmetros de *input* e no estado da aplicação. Esta camada controladora também é responsável pela selecção das páginas a visualizar, o que separa as JSPs e *servlets* em duas camadas distintas. As aplicações baseadas numa arquitectura *Model 2* apresentam um nível de manutenção reduzido e são mais fáceis de evoluir, pois não existem referências directas entre "vistas". A camada controladora fornece ainda um ponto único de controlo para segurança, *logging* e encapsulamento de dados de entrada de forma utilizável por um modelo *back-end* baseado em MVC. Por

[3]http://java.sun.com/javaee
[4]http://jade.tilab.com
[5]http://centria.di.fct.unl.pt/~jja/updates/

estas razões, este tipo de arquitectura é recomendado para a maior parte das aplicações interactivas, como é um caso de uma aplicação *web*.

A camada *model* representa toda a informação sobre o domínio de negócio. É uma camada não visual contendo todos os dados e comportamentos não usados pela interface do utilizador. Em vez desta camada ser implementada com simples classes de negócio, a presente solução propõe uma implementação orientada a agentes (ver figura 8.1). Este tipo de solução, à parte do dinamismo implícito dos agentes, apresenta diversas características interessantes e aplicáveis noutros paradigmas, tornando-a largamente reutilizável e integrável. Note-se que este núcleo bem definido pode ser integrado num ESB, sendo elevado a uma SOA.

A camada *view* apresenta a *model* na interface do utilizador. A *view* apenas apresenta informação, delegando à *controller* o tratamento das alterações de informação.

A camada *controller* recebe o *input* do utilizador, manipula a *model*, e faz com que a *view* se actualize de forma apropriada. Desta forma a interface do utilizador é uma combinação entre *view* e *controller*. Através do componente AgentServices (ver secção 8.3.4) a camada *controller* não conhece detalhes sobre a implementação usada pela *model*, estando o acesso a todos os agentes do sistema encapsulados nesta camada intermédia (ver figura 8.3). Este componente garante um nível de desacoplamento elevado no sistema.

Esta separação em camadas apresenta um conjunto de vantagens alargadas e é uma das heurísticas fundamentais no bom desenho de *software*. Note-se que a alteração da camada de negócio seria uma tarefa trivial, assim como é possível desenvolver múltiplas camadas de apresentação com a reutilização da camada *model*. Apenas se teria que respeitar a interface do componente AgentServices usada na *controller* para enviar o *input* para a *model*, e receber desta o *output* reenviando-o para a *view*.

Em suma, quando uma *servlet* (acção Struts) recebe o pedido do utilizador, envia-o para o UA e aguarda a sua resposta. Quando esta chega será redireccionada para uma JSP responsável por "renderizar" uma página HTML com conteúdos dinâmicos. O UA usa o motor de regras JLups programado para suportar alguns conceitos de uma arquitectura BDI em DLP. O motor de regras e exemplos de utilização estão descritos na secção 8.4.2 e 8.4.3, respectivamente.

Ao nível de segurança a aplicação usa autenticação baseada em *realms*, implementada através de JAAS. Com este tipo de solução é garantido um tipo de autenticação *pluggable*, ficando o código de negócio independente das particularidades de segurança. A autorização é baseadas em *roles* através da especificação de *Java Servlet 2.4*, que define a forma de estabelecer regras de controlo de acesso para restringir utilizadores aos vários recursos Java EE. Dois perfis de utilizadores foram criados e cada um está mapeado para um *role* Java EE, sendo que cada *role* tem acesso a um conjunto de recursos.

8.3 Arquitectura de Agentes

A arquitectura de agentes foi construída através de um modelo de três camadas [CAC04, pp. 67-68]. As funcionalidades de cada camada podem ser resumidas da seguinte forma:

- **Utilizador → Interacção com o Sistema.** Esta camada é responsável pela interacção com os utilizadores. Estes agentes (Agente do Utilizador ou Agentes de Interface) podem usar técnicas diferentes, tal como aprendizagem, comunicação entre utilizador e sistema.

- **Agentes do Utilizador → Agentes de Tarefas.** Camada que contém um conjunto de agentes especializados com um objectivo bem definido. Normalmente

são chamados de *Task Agents* embora sejam referidos por *Middle Agents*, *Execution Agents*, *Planning* ou *Learning Agents* noutras arquitecturas.

- **Agentes Intermédios → Agentes Web.** Esta camada engloba agentes que fornecem serviços, nomeadamente Agentes *Web*, *SoftBots*, *Crawlers*, entre outros que se especializam em recuperação e filtragem de informação da *web*, ou de outro repositório de dados.

De acordo com as camadas acima descritas a arquitectura de agentes (ver figura 8.2) foi composta por um conjunto de agentes por cada camada. Todos estes agentes colaboram com o objectivo de satisfazer os pedidos dos utilizadores.

8.3.1 Camada de Interacção com o Sistema

Esta camada contém um conjunto de agentes dedicados por exclusivo aos seus utilizadores.

Nela residem os agentes pessoais, um por cada utilizador no sistema. Este tipo de agente apresenta um comportamento particular que lhe permite aprender e evoluir no tempo. Quando "acorda", todos os factos aprendidos são disponibilizados - depois do carregamento da sua base de conhecimento. Após ser suspendido, toda a informação recolhida é armazenada para a próxima interacção.

Estes agentes mantêm ainda um comportamento similar a uma modelo BDI, onde gerem as suas crenças, controlam os seus desejos e executam as suas intenções. Estes agentes aprendem directamente através do seu utilizador ou através de outro agentes presentes no sistema. Por cada interacção, o agente verifica se existem novas intenções (derivadas de regras) para esse período no tempo. A sua base de conhecimento actualiza-se dinamicamente através do auxílio de um temporizador, resultando em possíveis novas intenções inferidas das suas regras.

As suas intenções são baseadas em pedidos do seu utilizador, portanto as regras de actualização da base de conhecimento são responsáveis pela gestão das intenções. Neste tipo de ambiente, o componente JLups (ver secção 8.4.2) forneceu a estes agentes a capacidade de crescimento, evolução, adaptabilidade ao ambiente e ao utilizador.

Cada um destes agentes, ou *User Agent* (UA), manifesta as seguintes características:

- Aprendizagem e adaptabilidade, no sentido em que aprendem através da interacção com o utilizador e melhoram o seu comportamento;
- Autonomia, pois podem escolher vários fluxos de execução para satisfazer os pedidos do utilizador;
- Cooperação, devido ao diálogo constante com outros agentes, principalmente serviços;
- Proactividade, pois informam o utilizador de novos resultados e mensagens, podendo eventualmente detectar intenções sem estímulos externo do utilizador;
- Continuidade Temporal, preservando o conhecimento adquirido em interacções anteriores.

Esta camada contém ainda os agentes que residem no ambiente mais próximo ao cliente (por exemplo, uma *applet*, uma sessão *web*, entre outros) que servem como canal de comunicação para a plataforma, e para os respectivos agentes principais do utilizador - implementando o padrão de desenho *Proxy* - herdando assim o nome de *Proxy Agent* (PA). Este tipo de agente está encapsulado no módulo AgentServices sendo o mais simples da plataforma, apenas mostrando indícios primários de comunicação, cooperação, e pura reactividade. Tal como o UA, existe um PA por utilizador.

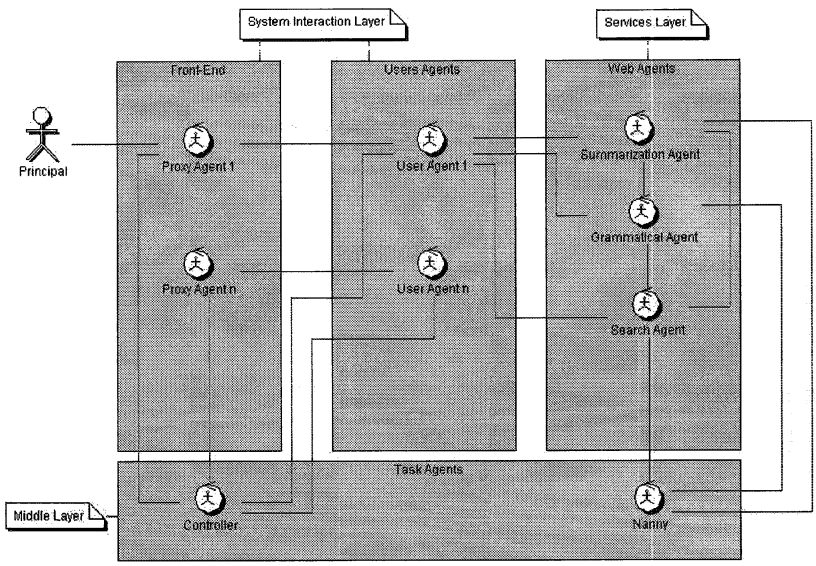

Figura 8.2: Architectura de Agentes.

8.3.2 Middle Layer

A camada intermédia (Middle Agents, *Task Agents*, entre outros) é composta pelos agentes de controlo. Nesta camada residem agentes únicos em todo o sistema que o controlam e o mantêm coerente.

O *Nanny Agent* (NA) controla o ciclo de vida de todos os agentes que fornecem serviços. Pode relançá-los, terminá-los em caso de erro ou ausência de resposta, registá-los no *Directory Facilitator* se necessário, gerindo também o *logging* do estado da plataforma. Como se foca apenas no controlo de qualidade do sistema pode ser visto como um mediador da plataforma. Contém um comportamento *threaded* que o permite controlar todos os serviços do sistema.

O NA manifesta as seguintes características:

- Autonomia, controlando o seu ciclo de vida;
- Comunicação com todos os agentes de serviço;
- Reactividade, pois executa tarefas se detectar determinados estímulos, como a morte de um serviço/agente.

Existe ainda um outro agente que controla o lançamento dos agentes na camada frontal, sendo nomeado do *Controller Agent* (CA). Por cada utilizador que entre no sistema o CA verifica se o UA respectivo já está presente no sistema e lança-o em caso negativo. Quando o utilizador sai do sistema o UA é hibernado e mantido num repositório para uso posterior, de forma a que todo o conhecimento adquirido seja preservado. O CA apresenta-se como um agente simples manifestando autonomia e um nível baixo de reactividade.

8.3.3 Services Layer

A camada de serviços contém três agentes *web*. O primeiro é o *Search Agent* (SA) que é responsável pela pesquisa de expressões ou palavras na *web*. Recebe um pedido de qualquer agente (normalmente um UA), pesquisa na *web* (neste caso através do *web service* disponibilizado pela Google) e devolve resultados. É possível especializar este

agente num outro contexto ou domínio particular, através da adição dinâmica de um novo comportamento[6]. A presente implementação é simples e apresenta-se como uma prova de conceito, deixando espaço para estudos posteriores com especialização de contexto.

O segundo serviço é fornecido pelo *Document Summarization Agent* (DSA) que é responsável pela sumarização de documento, recebendo pedidos de outros agentes para sumarizar determinados documentos. Apresenta-se como uma *façade* para o componente Sue [Pal03]. Este componente é um sistema Java de sumarização documentos, puramente extractivo e baseado no algoritmo descrito em [PRN02] [PRN03]. É composto por diversos módulos especializados na segmentação, *stemming*, eliminação de *stop words* e *ranking* (através do algoritmo TF-ISF ou por *keywords*). Este agente recebe um documento numa mensagem XML e retorna o *gist* e o sumário associado. Com este *gist* é possível refinar resultados com o auxílio de outros agentes. Contém ainda uma *façade* WSDL que o transforma num *web service*. Esta característica eleva a presente arquitectura a uma SOA moderna e viabiliza a integração deste sistema numa plataforma mais vasta, como um ESB.

O terceiro e último agente é o *Grammatical Agent* (GA) que contém uma base de conhecimento a nível gramatical, conhecendo definições, sinónimos, hiperónimos, hipónimos, entre outras. Recebe uma expressão ou palavra como pedido e devolve um conjunto de relações gramaticais como resultado, de forma a refinar de informação. Tal como os restantes agentes *web*, mantém um comportamento genérico e independente do contexto. A presente implementação usa a plataforma WordNet[7] como base de dados lexical para a língua inglesa.

Todos os agentes *web*, ou agentes de serviços, à parte do comportamento principal mantêm ainda um comportamento de *hearbeat* interpretado pelo NA. É através deste *hearbeat* enviado por cada agente que o NA controla e mantém a estabilidade da plataforma.

Os comportamentos principais dos agentes (pesquisas, apoio gramatical e sumarização) apresentam-se como *wrappers*, pois delegam a implementação das suas funcionalidades a serviços da camada de negócio (contidos no módulo susycore, ver figura 8.4). Este nível de indirecção torna a arquitectura mais flexível e escalável, assim como aumenta o nível de reutilização dos diversos componentes do sistema.

8.3.4 AgentServices

O módulo AgentServices (agentservices.jar na figura 8.4) foi construído com o objectivo de encapsular a tecnologia JADE e o seu *container* de agentes da aplicação *web*. Este módulo fornece um componente para iniciar, reiniciar e desligar o *container*; um componente para lançamento, pesquisa e recuperação de agentes, assim como o seu registo do *Directory Facilitator*. Todas as operações sobre agentes existentes na plataforma estão contidas neste módulo.

O *container* JADE é iniciado através deste módulo, invocado por uma *servlet* responsável pela inicialização da plataforma. Toda a comunicação entre acções Struts (*servlets*) e agentes é também feita através de serviços presentes neste módulo. Esta aproximação permitiu que o código de controlo (camada *controller*) permanecesse independente da tecnologia de agentes usada, viabilizando a alteração da API de agentes por uma outra compatível com as normas FIPA.

Em suma, este módulo pode ser visto como um *proxy* entre a camada *controller* e a camada *model*, presentes no padrão MVC (ver figura 8.3).

[6] JADE behaviour.

[7] http://wordnet.princeton.edu/

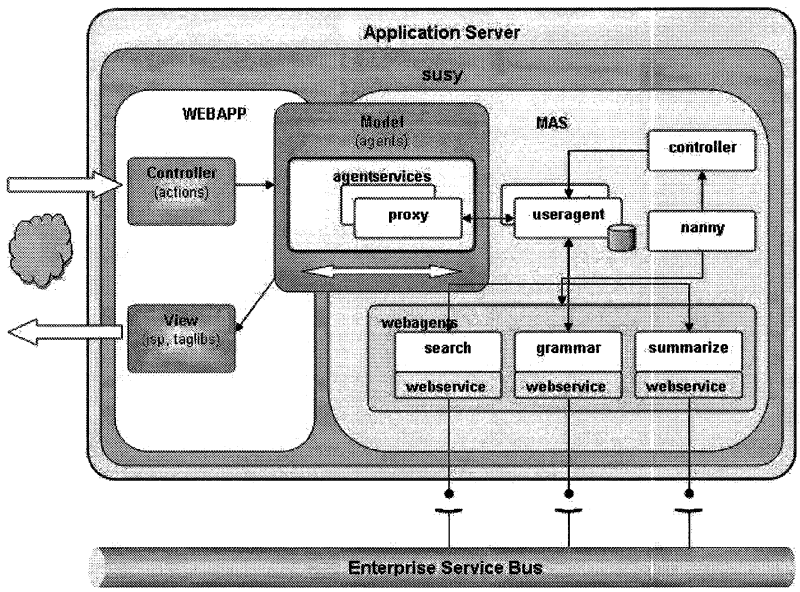

Figura 8.3: Susy, Vista Arquitectural.

8.4 Representação de Conhecimento

Os agentes pessoais dos utilizadores (UA) são os mais complexos e avançados no sistema. Usam a representação dinâmica de conhecimento para modelar o seu comportamento e estado. Através da DLP aprendem e evoluem no tempo. Cada acção do UA e a toda a sua lógica interna está modelada com esta tecnologia.

8.4.1 *Dynamic Knowledge Representation*

Um dos principais requisitos do formalismo usado para representar conhecimento é a capacidade de tratar e representar a sua evolução.

A DLP foi proposta por [ALP+98] como uma possível solução a este requisito evolutivo, e define como uma base de conhecimento pode ser actualizada por outra, obtendo uma nova base de conhecimento.

Especificamente, dada uma base de conhecimento original KB e uma outra de actualização KB', é possível obter uma nova base de conhecimento actualizada $KB^* = KB \oplus KB'$ que contém as actualizações de KB por KB'. De forma a criar um significado declarativo claro e facilmente verificável à actualização $KB \oplus KB'$, existe em [ALP+98] uma caracterização semântica completa da base de conhecimento actualizada $KB \oplus KB'$.

Para mais detalhes consultar Capítulo 3.

8.4.2 JLups

O JLups é um motor de regras Java para actualizações de conhecimento. Inspirado na linguagem LUPS [APP99b] [APP+99a] [ALP+98], o JLups é um projecto que simula o comportamento LUPS num paradigma diferente. Pode ser visto como um subconjunto da linguagem LUPS totalmente implementado na linguagem Java.

A LUPS como descrita acima, é uma linguagem declarativa para actualizações de conhecimento, que descreve transições entre estados de conhecimento consecutivos. É fornecida uma implementação em XSB-Prolog, no entanto como a interface XSB-Prolog com a linguagem Java é bastante primitiva, o uso desta implementação torna-se inade-

quada para adopção na arquitectura da presente tese. Por este motivo o componente JLups foi desenhado para simular o comportamento e algoritmos da linguagem LUPS, na plataforma Java.

O JLups contém associado uma interface de adaptadores de *input/output* que lhe permite interagir com diferentes plataformas e linguagens. É fornecido um adaptador para interpretar a sintaxe das regras LUPS[8], permitindo o carregamento que uma base de conhecimento LUPS possa ser importada e exportada. O JLups é ainda compatível com *Rule Markup Language* (RuleML)[9] através de um adaptador fornecido.

O núcleo deste componente é baseado em Mandarax[10], que é uma biblioteca Java *open source* para dedução de regras. A Mandarax fornece uma infraestrutura para definir, gerir e consultar regras e conhecimento. O facto de esta biblioteca ser baseada em *backward reasoning* torna-a coerente com a implementação base em XSB-Prolog, que usa o mesmo tipo de *reasoning*.

Este componente pode ser facilmente integrado com qualquer cliente Java, como uma *servlet*, um Agente, um EJB, uma *Applet*, entre outros. Na presente tese foi usado como base de conhecimento evolutiva dos agentes dos utilizadores (UA), fornecendo um comportamento baseado numa arquitectura BDI. Cada UA dispõe da sua base de conhecimento JLups, que evolui no tempo, permitindo gerir crenças e detectar intenções.

Para mais detalhes na arquitectura deste componente consultar Capítulo 7 e [PQ05].

8.4.3 Comportamento de Agentes

O comportamento dos agentes irá ser descrito através de simples exemplos codificados com regras LUPS e armazenados num repositório JLups. É importante referir que os agentes usam um componente para gestão de crenças e intenções que lhes oculta a noção de regras. Os exemplos que se descrevem abaixo omitem este componente dando a noção da existência de um canal directo entre agentes e JLups. Da mesma forma é omitida a camada de controlo (*servlets*) e a camada AgentServices - *proxy* entre o *controller* e *model*. Do ponto de vista arquitectural omitem-se estes fluxos para beneficiar o ponto de vista descritivo. Para um detalhe completo neste tipo de fluxo consultar o Capítulo 6, secção 6.4 Vista Lógica.

8.4.3.1 Pesquisa Automatizada I

Imagine que um utilizador do sistema deseja procurar informação sobre "mars". Depois de entrar no sistema e com o UA preparado para receber e tratar pedidos, o utilizador submete o *form* de pesquisa com o conteúdo especificado. Ao receber este pedido o UA processa-o através da aplicação da lei (8.1) que refere que cada pedido do utilizador é mapeado directamente para uma intenção do UA.

$$\begin{aligned} &always\ (int(ua, search(P))) \\ &\quad when\ (request(user, P)) \end{aligned} \tag{8.1}$$

Portanto, quando o UA consulta as intenções existentes na sua base de conhecimento (8.2) detecta uma consulta ao termo "mars":

$$holds\ int(ua, search(P)) \tag{8.2}$$

[8] A presente versão do JLups encontra-se ainda limitada na interpretação de regras complexas.
[9] http://www.ruleml.org
[10] http://mandarax.sourceforge.net

O UA recupera esta intenção e por reflexão invoca o método *search* com os paramêtros encontrados na variável P (neste caso P=["mars"]). O método de pesquisa envia uma mensagem ACL com a *performative* "REQUEST" ao SA. O SA invoca o Google e retorna um conjunto de resultados para o UA com uma mensagem ACL de *performative* "INFORM". Se algo correu mal, é usada a *performative* "EXCEPTION" dando origem a um tratamento do erro.

Quando o UA receber a informação proveniente do SA, irá armazená-la para ser posteriormente apresentada ao utilizador, finalizando a primeira fase deste fluxo.

A segunda fase começa após ser detectada a primeira intenção, através da regra (8.3), que refere que se há uma nova intenção de pesquisa e não se conhecem sinónimos para "mars" então pede-se apoio ao GA[11]. Um mecanismo de reflexão análogo ao anterior é usado, resultando na recepção de uma mensagem com pedido de informação sobre P, por parte do GA.

$$\begin{aligned} &always\ (int(ua, gram(P))) \\ &when\ (int(u, search(P), \\ &not\ bel(ua, synonym(P, Q))) \end{aligned} \tag{8.3}$$

O GA usa então o WordNet para satisfazer o pedido do UA. Recupera uma lista de sinónimos (ou hiperónimos e hipónimos) e envia-a de volta para o UA. Este por confiar incondicionalmente no GA, irá armazenar cada relação gramatical recebida na forma da regra (8.4).

$$assert\ (bel(ua, synonym(mars, roman\ god))) \tag{8.4}$$

Ao mesmo tempo que o UA receber informação do GA, irá também pedir ajuda ao SA na pesquisa de informação sobre os dados iniciais concatenados com os novos resultados vindos do GA - através das regras (8.5) e (8.6)

$$\begin{aligned} &always\ (int(ua, search("PQ")) \leftarrow \\ &int(ua, search(P)), \\ &bel(ua, synonym(P, Q))) \end{aligned} \tag{8.5}$$

$$\begin{aligned} &always\ (int(ua, search("PQ")) \leftarrow \\ &int(ua, search(P)), \\ &bel(ua, synonym(Q, P))) \end{aligned} \tag{8.6}$$

Quando o SA informa o UA com novos resultados estes irão ser adicionados à lista de resultados já existente na primeira fase, e enviados para visualização do utilizador.

Como descrito acima, o processo é realizado em duas fases. A primeira inicia-se aquando do pedido do utilizador e termina numa pesquisa simples. A segunda fase descreve um refinamento gramatical e o uso de crenças para enriquecer resultados. Cada uma destas fases corresponde a um estado na base de conhecimento.

[11]Note-se que se existisse conhecimento prévio sobre a variável em causa não iria ser pedido apoio ao GA.

8.4.3.2 Pesquisa Automatizada II

O utilizador irá neste exemplo pedir informação sobre "roman god". Ao submeter o pedido o UA irá detectar duas intenções derivadas do uso das regras (8.1) e (8.6). A consulta de intenções (8.2) irá retornar dois valores P = ["roman god"] and P = ["roman god mars"]. O segundo valor é derivado da regra (8.6) que obriga a uma pesquisa sobre P e Q se o utilizador pediu informações sobre P, sendo P um sinónimo de Q.

O UA irá posteriormente pedir a colaboração do SA para duas pesquisas, usando as crenças aprendidas anteriormente e evitando agora o uso do GA. Doravante, quando seja pedida informação sobre "roman god" a resposta do UA irá ser rápida e precisa com o uso de crenças.

Note-se que a qualidade desta resposta é mais interessante que a do exemplo anterior, pois através do uso da regra (8.3) o UA[12] irá pedir apoio ao GA para relações gramaticais sobre a expressão "roman god". Dos diversos resultados possíveis assume-se que "immortal" é retornado ao UA sendo assimilada como mais uma crença (8.7). Esta nova crença irá resultar numa nova pesquisa para os valores "roman god immortal".

$$assert\ (bel(gram, synonym(roman\ god, immortal))) \tag{8.7}$$

Através deste fluxo simples existe agora uma relação entre diversas expressões representada na base de conhecimento do UA (8.8).

$$\begin{gathered} bel(ua, synonym(mars, roman\ god)) \\ bel(ua, synonym(roman\ god, immortal)) \end{gathered} \tag{8.8}$$

Doravante qualquer pesquisa sobre "roman god" irá sempre retornar três resultados sobre as seguintes expressões:

- S1: "roman god";
- S2: "roman god immortal";
- S3: "roman god mars";

Note-se que os exemplos acima descritos não fazem distinção entre sinónimos, hiperónimos e hipónimos. São exemplo meramente ilustrativos com o objectivo de clarificar os fluxos existentes. É ainda importante referir que qualquer outra relação gramatical pode ser usada (p.e. antónimos). Na presente *release* a plataforma envia informação sobre três relações gramaticais anteriormente descritas.

É importante referir que as pesquisas *web* são auxiliadas com funcionalidades de *autocomplete*, através da tecnologia AJAX. Por cada letra inserida pelo utilizador na página de pesquisa, serão consultadas as crenças do agente e se existir uma correspondência entre palavra e crenças existentes, estas serão apresentadas no *browser* como sugestão de *autocomplete*. Para mais detalhe consultar Capítulo 6, secção 6.4.7.4.

8.5 *Deployment*

Esta secção descreve de forma resumida a vista de *deployment*, mostrando a distribuição física dos diversos componentes que integram a plataforma Susy (ver diagrama de *deployment* na figura 8.4).

[12] O UA nesta fase não tem nenhuma crença do tipo *bel(roman god,A)*. Apresenta sim uma crença *bel(mars,roman god)* o que faz com que a regra (8.3) seja válida.

- susyweb.war: a aplicação *web* contento a camada *controller* e *view*;
- susyagents.jar: contém os agentes JADE e os seus comportamentos;
- susycore.jar: toda a camada de negócio da aplicação está contida neste módulo, p.e. serviços usados pelos agentes;
- susysecurity.jar: contém o *realm* de segurança da aplicação *web*;
- agentservices.jar: contém serviços e classes utilitárias sobre agentes;
- jlups.jar: motor de regras com tecnologia DLP;
- sue.jar: sistema de sumarização de documentos.

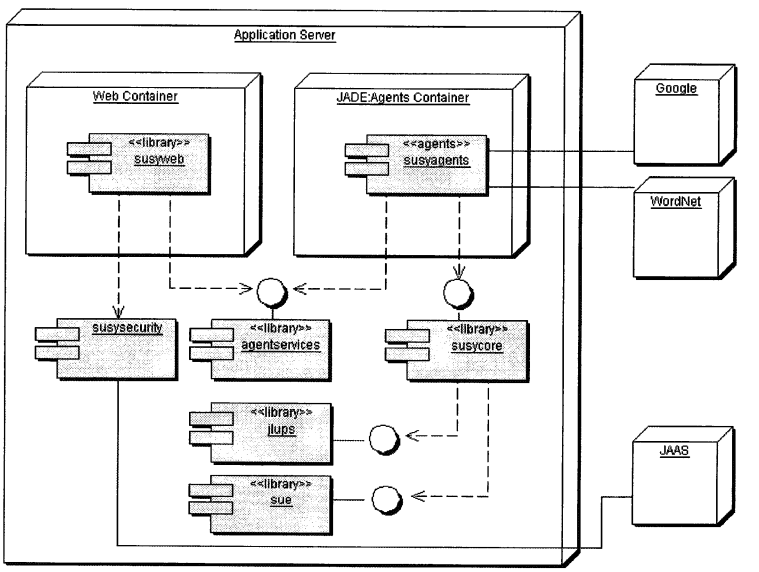

Figura 8.4: Diagrama de *Deployment*.

Parte III

Resultados e Conclusões

Capítulo 9

Testes

O presente capítulo analisa o desempenho dos módulos integrantes da arquitectura, assim como os componentes e agentes considerados mais significativos.

9.1 JLups

O JLups foi submetido a uma bateria de testes unitários, testes de integração e testes sistema. Nestes últimos foram ainda realizados testes de desempenho de forma a que a qualidade de funcionamento do componente seja interpretável.

9.1.1 Testes Unitários

O JLups foi submetido a um conjunto de testes unitários fornecidos juntamente com o seu código. Neste testes foram avaliados todos os componentes principais, tais como a fábrica de regras responsável pela criação de regras JLups - a partir de regras LUPS - e a implementação fornecida do JLups.

Os primeiros testes efectuados sobre a fábrica de regras evidenciam uma notória falta de "intuitividade" na criação das mesmas. A presente versão desta fábrica, que implementa uma interface bem definida, recorre ao uso de uma expressão regular para *parsing* das regras LUPS. A expressão regular, apesar da sua simplicidade mantém algumas limitações ao nível gramatical, não conseguindo esconder problemas actuais e futuros. Apresenta constrangimentos em pequenos detalhes, como por exemplo no número de espaços que separam alguns caracteres na regra, assim como não realiza o *parsing* de regras complexas. No entanto esta última limitação é um efeito secundário da presente versão e não da presente fábrica, pois a versão inicial do JLups foi desenhada para trabalhar apenas com regras simples. Como resultado final, este componente é uma solução simples para um problema complexo, e com determinados cuidados consegue fornecer alguma qualidade de serviço.

Os testes realizados à implementação fornecida para a interface JLups, ao nível unitário, foram desenvolvidos através da inserção de todos os tipos de regras na base de conhecimento, e da verificação de resultados. Desta forma foi igualmente testada a fábrica de regras acima descrita.

9.1.2 Testes de Integração e Sistema

Os testes de integração realizados à implementação do JLups foram mais complexos, e foram igualmente usados testes JUnit[1] para a sua realização. Para testar e provar o

[1]Disponíveis na *release* actual.

correcto funcionamento da base de conhecimento JLups foram usados dois programas LUPS:

- Programa "ex2.p"[2];
- Programa "ex3.p"[3].

Parte dos testes sistema foram realizados através do exemplo 2, enquanto a outra parte foi realizada num ambiente final da plataforma Susy, onde o JLups foi usado pelos agentes dos utilizadores (UAs) para armazenar e gerir conhecimento. Neste tipo de teste focam-se os valores comportamentais do JLups e focam-se também os diversos tipos de provas incidindo nos diversos tipos de regras com cláusulas *when*.

9.1.2.1 Testes com Exemplos LUPS (ex2)

O JLups foi testado exaustivamente através de um programa LUPS, exemplo 2 acima identificado. A figura 9.1 e 9.2 apresentam os tempos de cada operação realizada. Existem dez iterações que indicam que o exemplo 2 foi carregado dez vezes consecutivas, i.e. sem usar o comando *clear*. Com este fluxo é possível analisar-se o crescimento temporal das operações numa base de conhecimento JLups cada vez maior. As duas figuras apresentam o mesmo conjunto de resultados, no entanto visualmente tem perceptibilidades diferentes. A partir da primeira figura consegue notar-se uma ligeiro aumento no número de mili-segundos usados em cada operação, enquanto na segunda é possível notar que os comandos mais pesados são os *updates* e as *queries* (através do comando *holds*). O comando de *load* também foi analisado, apresentando o resultado temporal do carregamento de uma base de conhecimento do exemplo 2. Embora apresentando um valor elevado em relação às outras operações, serve apenas como indicador do tempo total consumido por carregamento.

É interessante notar que apenas os comandos de "*update*" são mais pesados. Como era expectável, as operações para adicionar e remover regras ou leis são as mais leves. Isto deve-se ao facto de não existir grande processamento aquando a introdução de uma operação deste tipo, pois as regras/leis são inseridas em listas que irão nos estados futuros ser avaliadas pelas actualizações.

9.1.2.2 Prova de Cláusulas *When*

O JLups usa quatro estratégias de prova[4] para regras com cláusulas *when*:

- A) Estratégia de Prova com Variáveis e NAF, versão "completa";
- B) Estratégia de Prova com Variáveis e NAF, versão "simplificada";
- C) Estratégia de Prova de Variáveis;
- B) Estratégia de Prova de Constantes.

A primeira estratégia (A) foi desenvolvida na fase inicial do projecto JLups, sendo substituída pela segunda (B) por motivos de desempenho. No entanto foi mantida no projecto por fornecer uma implementação completa que elimina o problema do *floundering*, pois quando existe uma variável livre num predicado com "*not*" esta estratégia constrói um mapa de todos os valores possíveis e prova a sua validade.

[2]Disponível em http://centria.di.fct.unl.pt/~jja/updates/xsbv/lups/ex2.p
[3]Disponível em http://centria.di.fct.unl.pt/~jja/updates/xsbv/lups/ex3.p
[4]Para mais detalhe consultar Capítulo 7, secção 7.3.1.

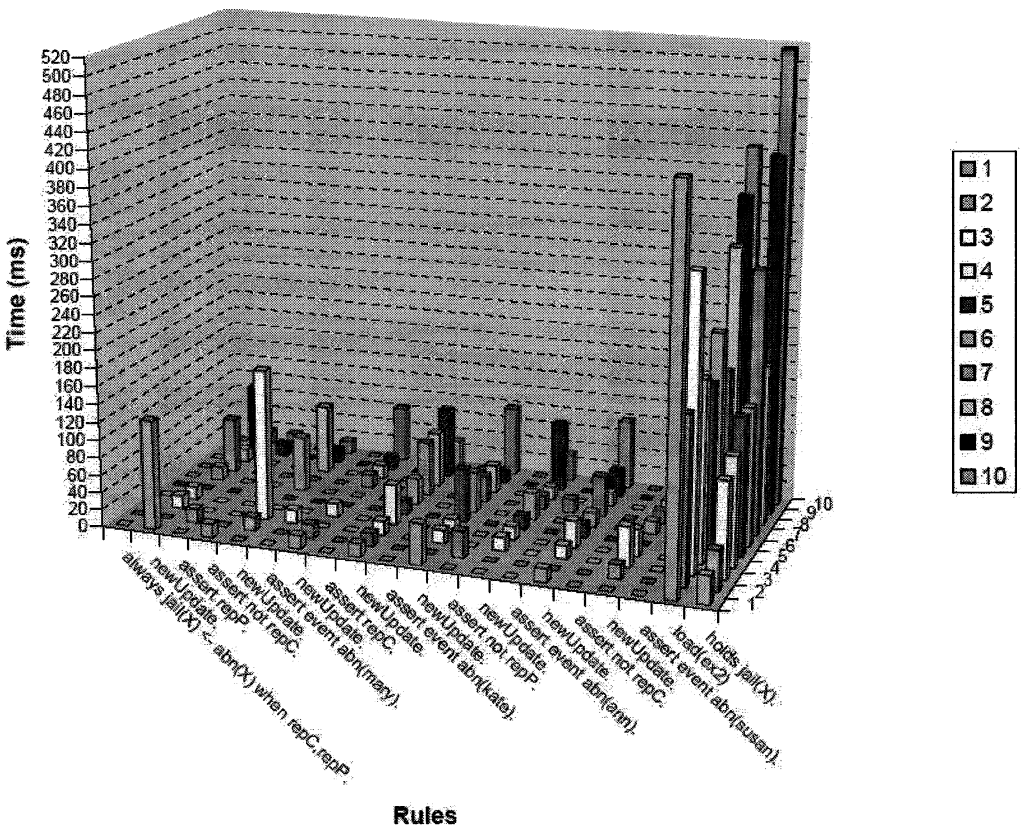

Figura 9.1: JLups - Testes de Desempenho ao Exemplo 2 (1).

Para ser perceber bem a diferença entre as duas estratégias, note-se o exemplo que se segue. Assume-se a existência da seguinte regra (entre muitas) numa base de conhecimento inicial de um agente do utilizador:

$$always\ event\ (int(gramm, R))\ when\ (request(automated, R), \\ not\ bel(synonym, R, S)) \tag{9.1}$$

Para este tipo de regras com NAF e variáveis (R e S) foi criada uma classe que verifica todas as combinações de valores possíveis dos dois predicados da cláusula *when*, e prova esses dois predicados com valores possíveis retirados da base de conhecimento. Esta implementação embora eliminando o *floundering*, pesa no sistema pois pode exigir muitas combinações entre valores de variáveis, e consequentemente muitas *queries* à base de conhecimento. No entanto o funcionamento é completo pois permite, depois de se encontrar todos os valores possíveis (através do exemplo da crença abaixo (9.2), com R='mars', S='red planet'), uma *query* com predicados NAF sem variáveis, que é o expectável mesmo em Prolog.

$$assert\ (bel("synonym","mars","redplanet")) \tag{9.2}$$

Existe portanto um problema bem conhecido com predicados NAF e variáveis. Esta classe desenvolvida é um mecanismo de prova para superar o problema de predicados NAF com variáveis livres, pois substitui todas as variáveis por valores existentes na base

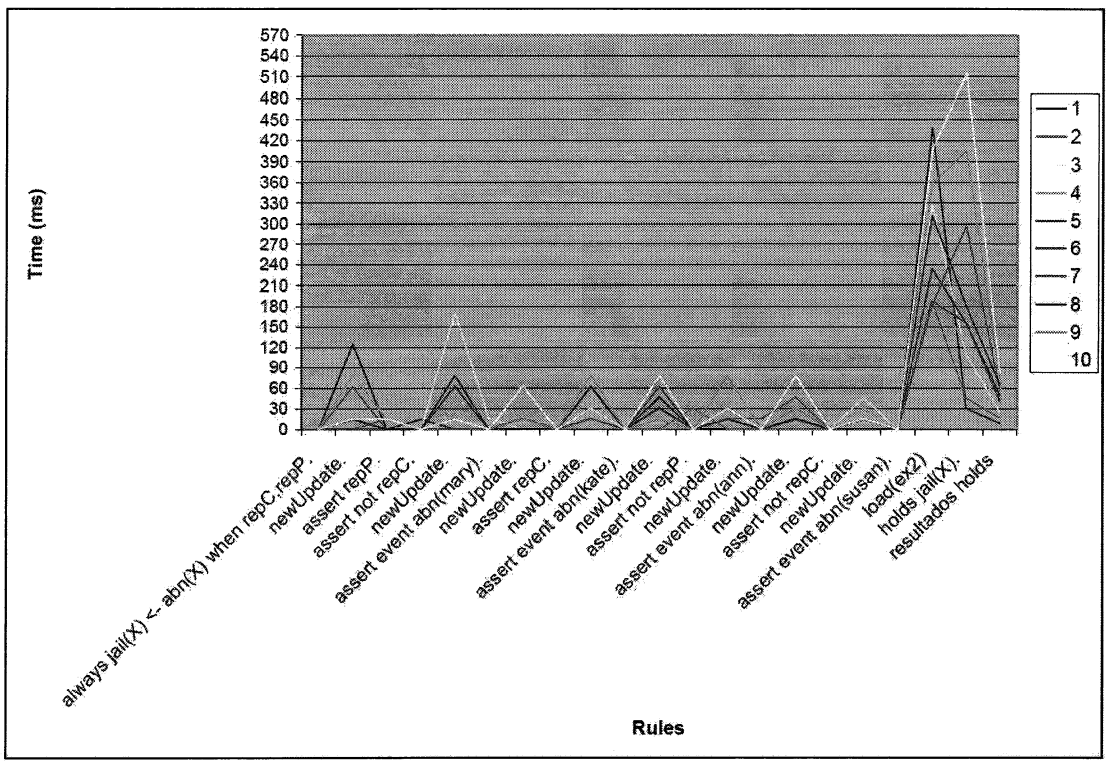

Figura 9.2: JLups - Testes de Desempenho ao Exemplo 2 (2).

de conhecimento permitindo a prova com NAF e variáveis nas cláusulas *when*.

O resultado dos testes de desempenho efectuados a esta estratégia revelam que esta implementação cria um *bottleneck* no sistema, pois consome muito tempo de prova (por quantas mais regras e estados existirem na base de conhecimento) e consequentemente demora muito na inicialização de um agente do utilizador e da sua respectiva base de conhecimento.

A segunda implementação (B) usa um algoritmo diferente e é uma derivação de um motor de inferência disponibilizado pela Mandarax, suportando prova de predicados com NAF e variáveis. Os resultados obtidos são interessantes:

Assume-se a existência da regra 9.1 na base de conhecimento e realizam-se os seguintes *asserts*:

$$assert\ (request(automated, mars)) \tag{9.3}$$

Ao usar esta nova implementação obtém-se o mesmo resultado - ou intenção - que na solução mais lenta e "completa":

$$int(gramm, mars) \tag{9.4}$$

No entanto se se alterasse a regra 9.1 para 9.5 a solução B não funcionaria, pois o S não está *grounded*, ou não está inicializado (pelo predicado *request(automated,R)*). Com a solução A é possível tratar a variável S como uma "intenção" pois tinha sido obtido um mapa de todas as possíveis combinações de valores. A solução B tem esta limitação, delegando algum trabalho ao programador DLP. No entanto mostra um elevado desempenho e apresenta um funcionamento correcto, pois se logo após a regra 9.6 se inserir a regra 9.3 não irão existir intenções de pesquisa sobre "mars" pois já existem crenças sobre o tema.

$$always\ event\ (int(gramm, S))\ when\ (request(automated, R), \\ not\ bel(synonym, R, S)) \tag{9.5}$$

$$assert\ (bel(synonym, mars, red\ planet)) \tag{9.6}$$

Estas duas soluções referidas são estratégias de prova (apresentando a função específica do padrão de desenho *Strategy*), e estão bem encapsuladas sob interfaces sendo apenas implementações que o JLups escolhe dinamicamente consoante o tipo de cláusula *when* a provar. A modificação de uma solução para outra é trivial, como a própria *pattern* transmite.

Como conclusão, a solução A embora completa, sofre de problemas de desempenho com o aumentar de predicados e estados numa base de conhecimento. O gráfico da figura 9.3 é exemplificativo desta situação, e mostra a solução B com um tempo de prova constante (40-50ms) enquanto a solução A vai aumentando os tempos de cada prova, com o crescer do número de predicados e estados.

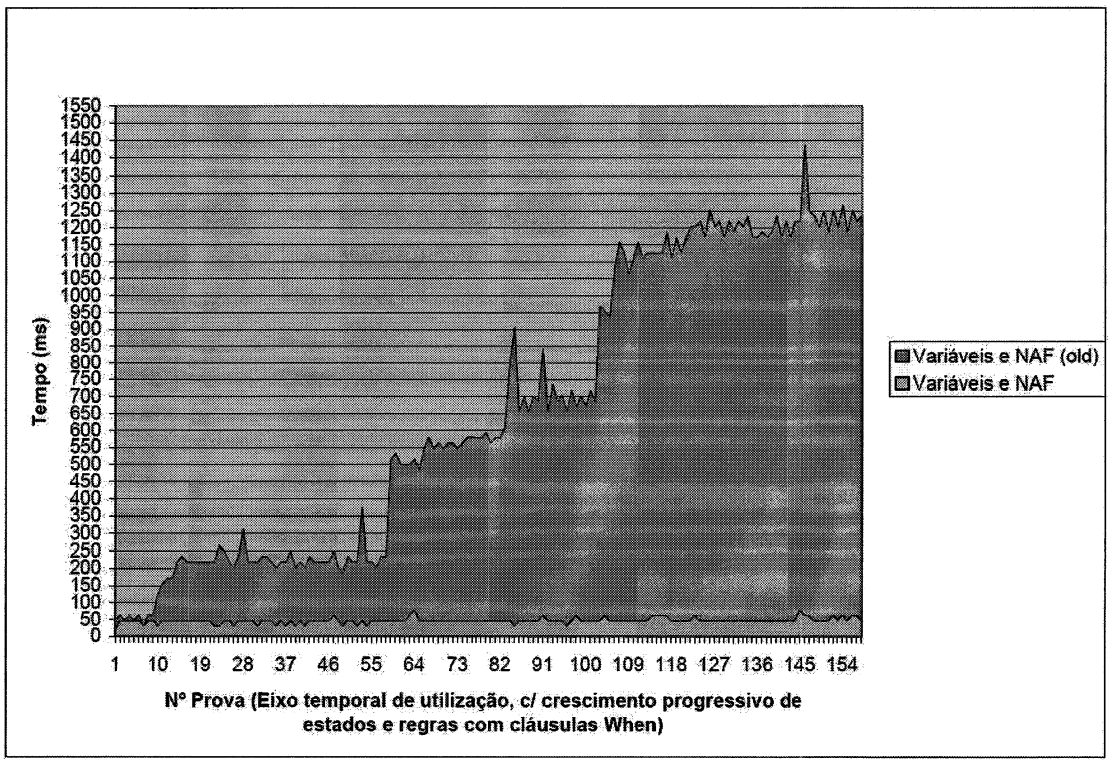

Figura 9.3: JLups - Prova de Cláusulas 'When', Comparação entre provas de predicados NAF.

As restantes estratégias de prova (C e D) de cláusulas *when* são mais simples, intuitivas e bastante mais rápidas. O desempenho de cada uma destas estratégias (juntamente com a solução B) pode ser analisada no gráfico da figura 9.4.

9.1.3 Testes sobre Estados e Crenças

Esta secção descreve treze testes realizados a várias bases de conhecimento, variando o número de crenças e o número de estados. Foram realizados em ambiente de qualidade, sendo bases de conhecimento integradas na plataforma Susy e usadas por agentes

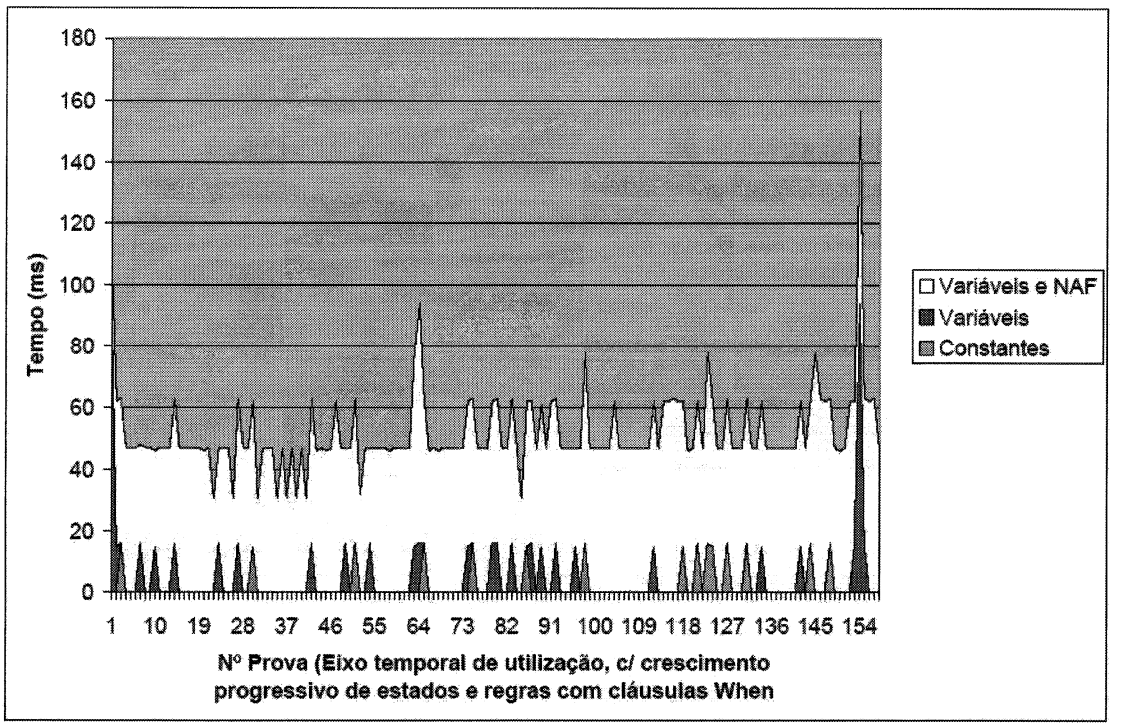

Figura 9.4: JLups - Prova de Cláusulas 'When', Comparação entre três tipos de prova.

pessoais.

Os primeiros testes contêm poucas crenças e estados que vão aumentando com a ordem dos testes. As figuras 9.5 e 9.6 apresentam os dados armazenados neste testes.

O primeiro teste foi realizado com uma base de conhecimento extremamente simples, apresentando trinta crenças e dois estados. Este teste difere do segundo no número de estados (o segundo contém cento e cinquenta e nove estados), sendo possível comparar os tempos de recuperação de dados e de carregamento entre um e outro. No décimo terceiro teste é possível perceber que estamos perante a base de conhecimento mais pesada, com cerca de duzentas crenças e quatrocentos e sessenta estados. Destes testes é notório um ligeiro aumento dos tempos de recuperação de dados (*query* às intenções e *query* às crenças) nos últimos testes, não obstante que esse aumento não é relevante em termos temporais nem afecta a escalabilidade do *software* pois está na ordem de poucos milissegundos. A dimensão também é reduzida para todos os testes, estando a maior base de conhecimento contida em poucos kilobytes (espaço físico no sistema de ficheiros). A ocupação de memória de uma base de conhecimento JLups passa quase desapercebida pois a gestão interna das regras é bem concretizada pela Mandarax. No entanto, a partir da figura 9.6 é possível comprovar que o número de estados influência largamente o carregamento duma base de conhecimento, podendo afectar o seu tempo de disponibilização.

9.2 AgentServices

Esta secção descreve os testes realizados ao módulo AgentServices, *middleware* responsável pela integração entre o MAS e aplicação *web*. São analisados os tempos consumidos na comunicação e em todas as colaborações presentes nas operações executadas.

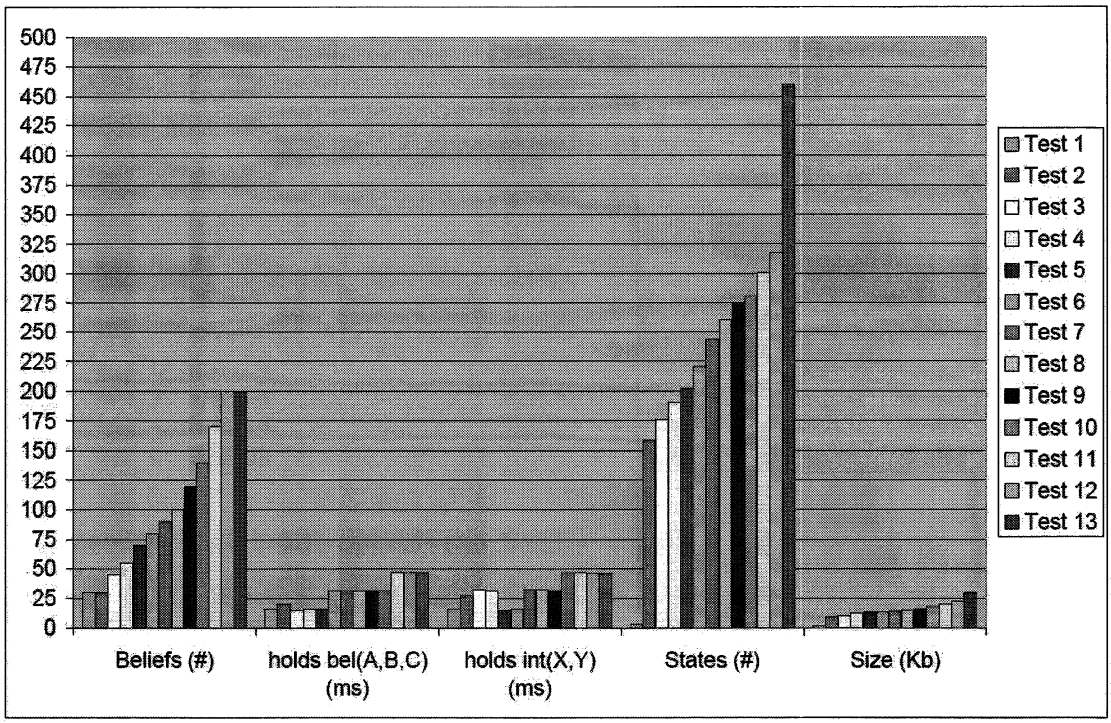

Figura 9.5: JLups - Testes sobre estados e crenças.

9.2.1 Listagem de Crenças

Foram realizadas quatro listagens em quatro bases de conhecimento diferentes, cada uma com determinado número de estados e com determinado número de crenças. A figura 9.7 apresenta as condições das listagem, assim como o tempo da query (*holds*) e tempo total que demorou a recuperação de dados para visualização em HTML.

A base de conhecimento da primeira listagem apresenta cerca de duzentas crenças e aproximadamente seiscentos estados. A segunda apresenta as mesmas crenças que a primeira embora com apenas cerca de quarenta estados. A terceira apresenta o mesmo número de estados da segunda embora com cerca de vinte crenças. A quarta e última listagem incide sobre uma base de conhecimento com as mesmas crenças que a anterior, no entanto apresenta os mesmos estados que a primeira listagem.

A partir da figura 9.7 é possível notar que a primeira listagem é a mais pesada, tanto em tempo total como em tempo de *query*. Note-se que o número de crenças afecta directamente o tempo de recuperação de dados, em detrimento do número de estados que não tem influencia relevante nas listagens. Este tempo acrescido, em bases de conhecimento com muitas crenças é expectável, visto nas duas primeiras listagens passar um conjunto de dados muito maior que nas restantes. O custo deste fluxo de dados é mais notório nos tempos totais que nos tempos de *query*. Os tempos consumidos nas *queries* são muito semelhantes entre listagem, no entanto as duas últimas por serem mais leves em termos de conteúdos apresentam tempos ligeiramente mais reduzidos.

9.2.2 Pesquisa Automatizada Com Crenças

Os testes à funcionalidade de pesquisa "automatizada" foram realizados da mesma forma que os anteriores. Foram lançados quatro testes a pesquisas, cada um contendo uma crença directa para pesquisa (resultando em duas pesquisas pedidas ao SA, uma simples e outra usando a crença existente), com a sua base de conhecimento contendo determinado

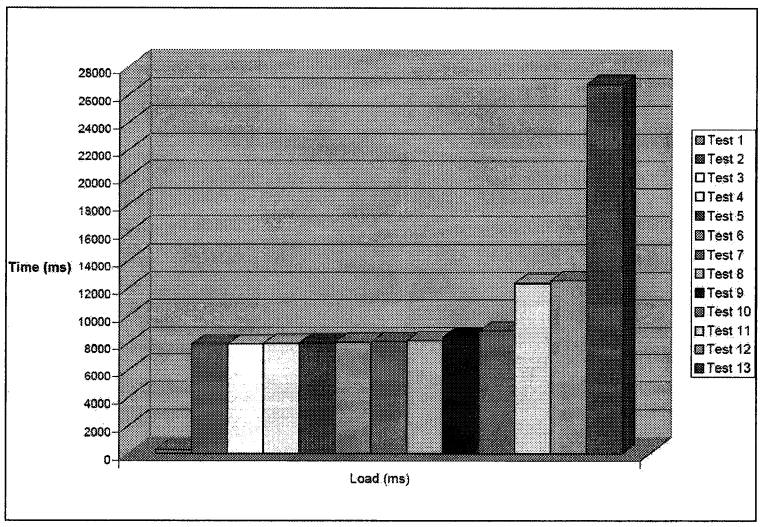

Figura 9.6: JLups - Tempos de Carregamentos dos Testes.

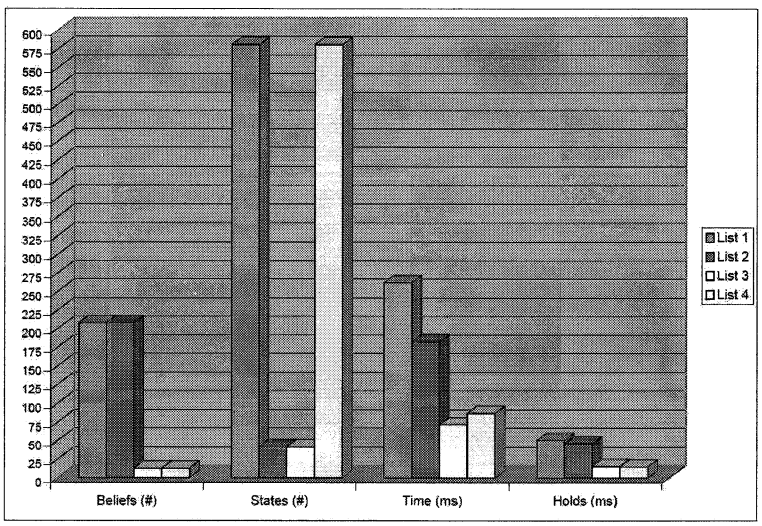

Figura 9.7: AgentServices - Listagem de Crenças.

número de estados e crenças (ver figura 9.8). Nestes testes foram analisados os seguintes tempos[5]:

1. Tempos totais consumidos pelo agente do utilizador (UA) desde o pedido do utilizador, até a resposta;
2. Tempos para adicionar o pedidos do utilizador (no UA);
3. Tempos para recuperar as intenções (UA);
4. Tempos parciais de pedido de pesquisas ao agente de pesquisa (SA), desde o pedido feito pelo UA até à resposta do SA;
5. Tempos consumidos por cada invocação ao *web service* de acesso ao Google, e o tempo total usado pelo SA.
6. Tempos totais usados pelo SA;

[5] A começar no terceiro conjunto de resultados no gráfico da figura 9.8, respectivamente.

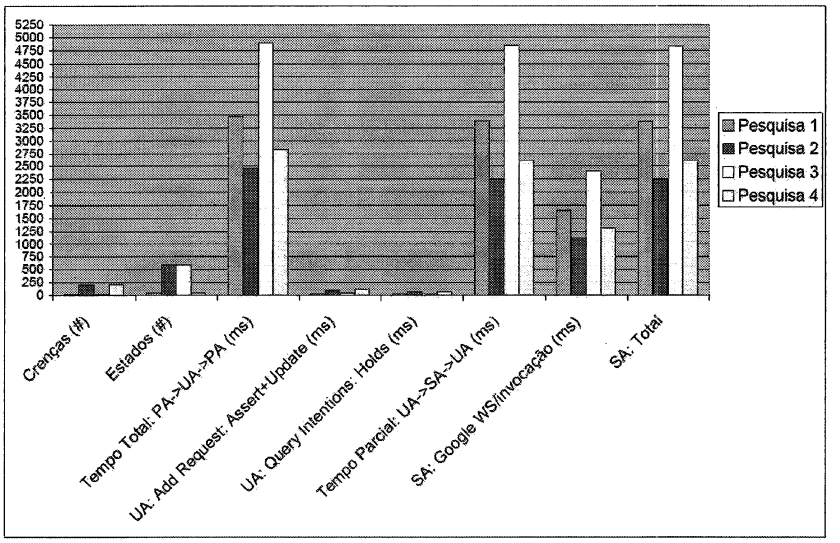

Figura 9.8: AgentServices - Pesquisa Automatizada Com Crenças.

A partir dos valores presente no gráfico da figura 9.8, pode-se concluir que os tempos consumidos no uso do JLups (2 e 3) são muito reduzidos, apresentando valores na ordem dos poucos milisegundos e confirmando os valores do gráfico da figura 9.5.

Conclui-se também que os tempos totais de uma pesquisa automatizada variam bastante (1), devido à pouca disponibilidade do *web service* da Google, e devido também à reduzida eficiência do mesmo. Note-se que o tempo total usado pelo SA (6) é na invocação deste *web service* e na espera por resultados. Se se multiplicar por dois o tempo de cada invocação ao *web service* da Google (5), irá obter-se um valor muito semelhante ao total usado pelo SA (6), o que indica que o SA consome o seu tempo neste processo. Seguindo este raciocínio e comparando este último valor (6) com o tempo total consumido pelo UA (1), verifica-se que a qualidade de resultados fica directamente influenciada por os valores associados à invocação do *web service* externo. É ainda possível verificar que o "custo" de usar mensagens e fluxos assíncronos para processar a lógica de negócio é irrelevante, pois cada troca de mensagem é realizada em poucos milisegundos[6].

[6]Pode-se retirar esta conclusão subtraindo aos valores de (1) os valores de (6).

Capítulo 10

Conclusões

Este capítulo conclui a presente dissertação, retirando conclusões, avaliando futuros trabalhos e próximas alterações ao projecto.

10.1 Notas Arquitecturais

10.1.1 Estilo

Tecnicamente, o maior risco inicial identificado foi a integração de uma plataforma de agentes com uma plataforma orientada para a *web*. Outro risco claramente elevado, foi a utilização de tecnologia DLP para modelar a lógica de alguns agentes. Como nota geral e sucinta sobre o estilo arquitectural seguido na presente dissertação (ver secção 6.2.3 Estilo Arquitectural), pode ser constatado um balanço positivo.

A presente dissertação, acompanhada da sua plataforma de *software*, demonstra como de forma harmoniosa se podem integrar tecnologias tão diferentes cooperando para o mesmo fim. Este passo arquitectural deve ser interpretado como uma proposta menos convencional, para atingir o mesmo que fim que outras soluções com tecnologias mais *standards*, promovendo a área de inteligência artificial para o efeito.

Um sistema multi-agente bem desenhado, bem implementado e testado, pode ser usado para fornecer a lógica de negócio a um sistema com maior dimensão e mais complexo. Esta tese demonstrou ainda a facilidade de como um agente pode apoiar o negócio de uma aplicação, e como pode ainda ser elevado a um serviço, podendo esta arquitectura ser promovida a uma SOA sem necessidade de redesenhar as bases.

Esta dissertação revela ainda cuidados e passos importantes a realizar, exigindo conhecimento teórico e prático nas diversas áreas focadas, desde a área de desenho até à área de desenvolvimento - podendo sem dúvida exigir algum potencial técnico às equipas de "terreno". Não obstante, este estilo em nada difere de outro estilo na curva de aprendizagem, pois as tecnologias base são em grande parte comuns a outros estilos arquitecturais.

Os testes unitários, os testes de integração e os de sistema comprovam ainda a boa usabilidade da solução. Estes testes também indicam pontos a melhorar, sendo que não põem em causa os estilos mas sim algumas limitações técnicas e pontuais por resolver.

Como nota final ao estilo seguido, é importante reiterar que a utilização do padrão MVC concretizou o sucesso inicial pretendido, promovendo a correcta e essencial separação destas camadas e estilos, sendo premissa fundamental para o bom desenho da solução.

10.1.2 Fluxos Assíncronos

As tecnologias assíncronas, via *web* e comummente apelidadas de AJAX estão a conquistar não só os utilizadores convencionais, mas também os arquitectos e *developers* da área. Esta dissertação apresenta vários exemplos de comunicação assíncrona, implementada com ferramentas actuais, que demonstram como este tipo de tecnologia pode ser utilizado em projectos Java EE.

Para o utilizador, esta tecnologia fornece um estilo de utilização e interacção mais elevado e nada comparado ao mais antigo estilo clássico e síncrono. Fornece uma outra dimensão ao nível de utilização e usabilidade.

Para arquitectos e *developers* é importante referir que este tipo de tecnologia promove tempos de desenvolvimento mais curtos, retirando complexidade de desenho e de detalhe técnico nos fluxos, não quebrando o modelo MVC. As típicas soluções com MVC existentes e utilizadas na indústria (p.e. Struts), embora interessantes, não escondem os largos tempos de desenvolvimento implícitos a um projecto que implique estas características. Existe claramente um preço a pagar ao nível da qualificação de quem desenvolve com este estilo, portanto um preço inato pela qualidade quando se inicia o processo de desenvolvimento de *software*.

Por muito que as características humanas de resistência à mudança sejam difíceis de ultrapassar, este caminho técnico apresenta-se como uma solução e como um dever de ser explorado. A utilização desta tecnologia com as actuais e futuras APIs, visa facilitar todo o processo de desenvolvimento assim como melhorar a interacção com o o utilizador. Esta tese, mais do que apresentar exemplos de como desenhar, modelar e desenvolver com esta tecnologia, apresenta também o caminho para a integrar com um sistema multi-agente[1] de forma simplificada. A ligação entre AJAX e *messaging* para agentes é uma solução interessante e coerente que merece ser analisada em maior detalhe.

10.1.3 Sistema Multi-Agente

O sistema multi-agente descrito na secção 6.3.3 prevê três camadas de agentes. Dentro destas camadas prevê ainda agentes com diferentes perfis. Após este sistema estar em funcionamento e ter sido sujeito a vários testes unitários e de integração com a plataforma *web*, pode concluir-se que o modelo escolhido é funcionalmente aceitável e bastante estável.

Este modelo apresenta uma simplicidade funcional inata, facilmente perceptível na secção supra-mencionada. Note-se que pela sua simplificada forma "básica" proposta, este modelo permite que em fases posteriores do projecto sejam adicionados mais agentes assim como novas camadas.

10.1.3.1 NA - *Nanny Agent*

Dadas as características da plataforma desenvolvida, que por ser *web* é naturalmente orientada para a interacção do utilizador, é importante o leitor notar a vital importância do agente NA nesta infra-estrutura. Com este agente o MAS fica controlado ao nível de serviços, qualidade de serviço e disponibilidade, podendo eventualmente estimular outros agentes que interajam com a componente *web*. A existência de um agente com estas características numa arquitectura composta deste tipo, é uma peça vital para a sua estabilidade.

[1] É importante recordar que toda a arquitectura de agentes funciona através *messaging*, sendo um solução de comunicação assíncrona por definição.

Dada a importância deste agente, deve ser ponderado o desenho de mais regras de forma a que este proceda à execução de determinadas tarefas, com base num conjunto de estímulos mais complexo. Estas regras podem inclusive ser introduzidas numa base de conhecimento JLups, sendo outro objecto de avaliação ao nível da tecnologia DLP.

Em suma, este agente fornece de forma clara um motivo de reflexão sobre o seu objectivo e papel num sistema com estas características. A sua funcionalidade, essa, é inquestionável.

10.1.3.2 CA - *Controller Agent*

O agente CA desempenha um papel simples e pouco interessante na plataforma. Deverá ser sujeito a uma avaliação ao nível funcional, podendo abarcar um novo conjunto de funcionalidades que fortemente justifiquem a sua presença na plataforma, ou que justifiquem a sua eliminação por não trazer grandes vantagens na sua existência.

10.1.3.3 PA - *Proxy Agent*

O agente PA é um agente simples mas essencial para a boa separação de camadas. Reside num espaço entre a camada *web* e o MAS, funcionando como canal de comunicação entre ambas. A sua existência numa plataforma com esta separação de camadas é claramente justificada, podendo eventualmente crescer do ponto de vista da complexidade ao nível de serviço de *proxy*.

Note o leitor que este agente está encapsulado no módulo AgentServices, estando fisicamente isolado dos restantes agentes. A sua reutilização é claramente elevada.

10.1.3.4 UA - *User Agent*

O agente UA é o mais complexo do sistema e tem foco especial neste estudo, por introduzir tecnologia DLP no seu "raciocínio". Através dos testes realizados no capítulo anterior é possível afirmar que a introdução desta solução foi bastante positiva, não só tecnicamente e ao nível de desempenho, mas também por abrir várias portas para futuras arquitecturas, desenhos, e desenvolvimento.

Foi ainda possível comprovar que a comunicação do UA com o sua unidade de memória lógica[2] está bem desenhada do ponto de vista estrutural, pois existe uma camada intermédia que oculta o paradigma DLP e JLups ao UA. No entanto esta camada sofre de algumas limitações ao nível da interface, que deverão ser revistas ou resolvidas aquando da introdução de uma arquitectura BDI pura. Estas limitações incidem sobre os tipos de *input* e *output* desta camada intermédia, que deverão ser apurados por um modelo de mensagens BDI concreto.

Na comunicação entre este agente e os restantes agentes de serviços não foram detectados problemas. Quando uma resposta excede o tempo pré-definido[3] este agente gera uma notificação para o PA, de forma a que este controle a componente *web* para eventuais problemas. Sempre que uma resposta chegue atrasada esta não é ignorada, sendo passada ao PA para tratamento. Desta forma o utilizador não fica à espera de uma resposta do negócio que nunca chega, e pode sempre consultar resultados "fora de tempo".

Dada a complexidade deste agente, é convidativo repensar a sua estrutura interna para que o tratamento das suas intenções seja realizado de uma forma mais parame-

[2]Neste caso o JLups, no entanto com estas camadas definidas torna-se fácil alterar a implementação desta funcionalidade.

[3]Por exemplo, quando o SA aguarda demasiado tempo pela resposta do serviço da Google.

trizável. Isto poderá implicar que no futuro exista um ficheiro de meta-informação que auxilie a parametrização das acções concretas a desencadear pelo agente.

10.1.3.5 SA - *Search Agent*

O SA é um agente de serviço bem identificado e desenhado. No entanto sofre de um problema ao nível da implementação por usar um *web service* da Google para pesquisas, que mantém um desvio padrão entre respostas considerável. Não sendo um problema directo sobre este agente, esta situação afecta claramente o desempenho no retorno de dados. No futuro poderá usar um mecanismo de *cache* e até uma base de conhecimento JLups para tornar a recuperação de informação mais rápida. Esta implementação poderá ser revista ou alterada, sendo que a interface de comunicação se encontra bem definida.

10.1.3.6 GA - *Grammatical Agent*

O GA, analogamente ao anterior é um agente bem identificado, que deverá ser revisto do ponto de vista de conteúdo de mensagem. As suas mensagens deverão ser mais complexas, mantendo uma estrutura gramatical completa e semanticamente correcta, sendo o mínimo exigido num agente com estas características. A presente implementação é simples e serve como prova de conceito, usando o WordNet para recuperar informação gramatical. Esta solução poderá prevalecer sendo que deverá ser reestruturada para enviar mais detalhe sobre o "tema" pretendido.

10.1.3.7 DSA - *Document Summarization Agent*

O DSA é o último dos agentes de serviços, e tal como o SA é um agente bem identificado e desenhado, usando como implementação um componente de sumarização extractivo de documentos - Sue. Ao nível de desempenho herda a boa qualidade da biblioteca Sue, no entanto deverá expor as parametrizações de cada sumarização na sua interface de comunicação e de serviço, de forma a crescer na qualidade funcional.

10.1.4 DLP e JLups

O componente JLups exerce um papel importante na presente arquitectura, fornecendo aos agentes uma forma de estes atingirem os objectivos através de um motor de regras dinâmico e evolutivo. Por outras palavras, fornece uma forma para estes agentes "raciocinarem". Desta forma, em detrimento dum modelo estático para codificação do núcleo dos UAs, optou-se por um modelo dinâmico, evolutivo no tempo, e orientado a regras. Através do JLups estes agentes conhecem o conceito de tempo e estado, do ponto de vista do seu conhecimento interno, podendo ter codificadas regras que se mutam, que alteram todo o estado interno inserindo novas regras ou removendo outras antigas.

Os testes efectuados no capítulo 9 (secção 9.1) a este componente são representativos do estado actual do desenvolvimento. É um componente bem identificado que requer algumas correcções e melhorias, de forma a ser mais robusto e utilizável. Os testes evidenciaram um *parser* de regras algo primitivo, assim como algumas limitações ao nível funcional. Do ponto de vista de desempenho a solução é bastante rápida e permite lidar com um certo volume de dados de forma eficiente.

Ao nível das estratégias de prova é importante referir que o facto de este comportamento estar separado do objecto que o gere, permite uma alteração dinâmica da estratégia de prova a usar. Existem várias estratégias de prova para vários tipos de regras, no entanto a prova de regras com claúsulas NAF merece maior foco por existirem duas soluções, uma completa e outra rápida, como características principais. A

implementação completa elimina o problema do *floundering*, pois quando existe uma variável livre num predicado com "*not*" esta estratégia constrói um mapa de todos os valores possíveis e prova a sua validade. A outra solução é mais rápida e não elimina o problema do *floundering*, sendo o programador o responsável pelo correcto desenho das regras.

A utilização deste componente torna interessante o agente UA, pois abre caminho para a utilização de tecnologia DLP em plataformas de maior dimensão.

Parte do objectivo desta dissertação era também avaliar o potencial das tecnologias DLP em infraestruturas de *software* reais. O uso desta tecnologia está embebida no JLups, que se apresenta no inicio da sua vida e se apresenta também como uma alternativa *opensource* a outro tipo de solução lógica, mesmo dentro do mundo dos motores de regras - proprietários ou não. Por exemplo, o motor de regras Jess[4] embora interessante, tem um formato proprietário, não sendo *opensource* e apresentando um custo para fins comerciais. Mesmo sendo uma solução a ponderar para núcleo de um agente, está longe de conseguir enquadrar-se no paradigma e potencial fornecido pelas tecnologias DLP.

10.1.5 Reutilização

O desenho e desenvolvimento do *software* que acompanha a presente dissertação, foi orientado à reutilização de componentes, desde um ponto de vista macro (módulo de *software*) até ao nível da classe.

Do ponto de vista funcional interessa pôr foco na Camada Genérica, descrita na secção 6.4.6.3, com o módulo AgentServices, e no Capítulo 7 que descreve o módulo JLups. Estes dois módulos apresentam uma natural e elevada taxa de reutilização, por serem soluções independentes do domínio. Note-se que o módulo AgentService poderá integrar qualquer plataforma como bloco de *middleware* responsável pela comunicação com um MAS. O JLups é um componente DLP que pode integrar o núcleo lógico de qualquer objecto que deseje utilizar tecnologia DLP.

Ao nível da Camada Específica do Domínio (ver secção 6.4.6.2), o módulo susycore apresenta-se como uma pequena biblioteca ou *framework* para desenvolvimento de agentes e serviços, sendo reutilizável numa aplicação com um contexto semelhante, ou até numa solução *agent-oriented* mais genérica[5].

Ao nível da reutilização do MAS, é importante referir que este sistema está disponível para ser integrado numa outra aplicação, sendo que nesta dissertação é usado conjuntamente com uma plataforma *web*. Os agentes que pertencem e criam este MAS também podem ser reutilizados para outros fins, dadas a suas características independentes de desenho. A camada de serviço no MAS será a que apresenta o maior nível de reutilização, sendo que a camada de controlo também poderá ser reutilizada como agentes de controlo genéricos duma plataforma de agentes. O UA pode efectivamente ser extraído desde MAS e ser reutilizado em aplicações que exigem uma forte interacção entre sistema e utilizador. Ao usar-se o módulo AgentServices, o PA é implicitamente reutilizado.

É ainda importante referir que a correcta identificação de agentes (ver secção 6.3.3) no processo de desenho de *software* se mostrou essencial para a qualidade, manutenção, evolução e reutilização de todo o MAS.

[4]Ver http://herzberg.ca.sandia.gov/jess/.

[5]Por introduzir um conjunto de classes que fornecem uma base alargada de funcionalidades, assim como herda características de desenho que já resolvem vários tipos de problemas comuns num MAS.

10.1.6 Orientação a Serviços

Mais que uma arquitectura de agentes e uma arquitectura *web*, a presente dissertação propõe a elevação da presente solução a uma SOA. Através do processo de identificação de agentes (ver secção 6.3.3) complementado com o processo de identificação de serviços (secção 6.3.2), introduzido pelo RUP, foi possível construir uma arquitectura de agentes com diversas camadas, existindo uma dedicada ao puro fornecimento de serviços - facilmente traduzível numa SOA.

O facto de os agentes da camada de serviços, ou *Web Agents*, estarem bem definidos, estando o seu natural objectivo mapeado para um serviço, torna toda esta infraestrutura reutilizável num escopo mais alargado, podendo com facilidade ser integrada num *bus* de informação. A partir deste desenho e desenvolvimento, é possível notar a agilidade de uma plataforma de agentes, podendo ser utilizada para vários fins e colaborando com diversas outras tecnologias, em ambientes heterogéneos e dispersos.

Como nota final, é importante referir que uma solução alternativa a expor os agentes como *web services*, seria desenvolver um adaptador JCA genérico de forma a que estes agentes pudessem ser adaptados directamente a um ESB.

10.2 Trabalho Futuro

A plataforma Susy tal como é actualmente, uma arquitectura *web* conjunta com uma arquitectura de agentes, fornece um conjunto de funcionalidades diversificadas. É uma plataforma aberta, pois esta dissertação apresenta todo o detalhe técnico para que se mantenha a evolução do sistema. Algumas funcionalidades poderão ser adicionadas, outras redesenhadas, para que seja assegurada uma contínua evolução da plataforma. Abaixo seguem pontos que deverão ser revistos nas próximas etapas do projecto, servindo de resumo funcional sobre a secção anterior:

- Mais Funcionalidades, Mais Agentes: Deverão ser equacionados novos agentes de serviços, que poderão complementar a plataforma e introduzir uma maior grau de interacção entre utilizador e respectivo UA. Dada a natureza da plataforma serão equacionados serviços orientados à recuperação de informação;

- Revisão de Agentes de Serviços: O DSA deverá expor na sua interface de comunicação as parametrizações de sumarização da biblioteca Sue. A implementação de pesquisa usada pelo SA deverá ser reequacionada. A interface de comunicação com o GA deverá conter um *output* mais complexo e com informação gramatical elaborada sobre um pedido;

- Segurança e Maior Controlo da Plataforma: Numa futura versão a plataforma Susy apresentará comunicação segura entre agentes. Da mesma forma poderão existir mais agentes controladores, como p.e o agente de notificações, apresentando uma forte interacção com um utilizador administrador. O NA deverá crescer em número de funcionalidade, de forma a poder controlar com maior segurança a qualidade de serviços da plataforma. Novas regras introduzidas ao NA devem passar por um estudo que analise se este agente deverá ser elevado ao nível de "inteligência", podendo para o efeito ter uma base de conhecimento JLups para o auxiliar no processo de decisão;

- Mobilidade de Agentes: Todos os UAs poderão ser descarregados e carregados pelo seu utilizador, assim como poderão apresentar mobilidade entre plataformas JADE;

- Agentes, Regras e Execução: Separação do código de execução de regras dos comportamentos principais do UA;
- Eliminação de Fluxos Síncronos: Generalização de todos os fluxos *web* através de invocações assíncronas com AJAX;
- Arquitectura BDI: Estudo de uma possível evolução para uma arquitectura BDI pura ao nível do UA;
- Integração num ESB: Integração e disponibilização dos agentes de serviços numa SOA através de um ESB[6];
- *Realm* JAAS baseado em agentes: Poderá ser considerado o desenvolvimento de um *realm* para autenticação Java EE totalmente baseado em agentes, tal como existem para várias outras tecnologias (p.e. ficheiro, JDBC, LDAP, entre outros).

Ao nível do componente JLups está previsto um conjunto interessante de tarefas a realizar, num âmbito totalmente diferente dos pontos anteriormente descritos:

- Desenvolvimento de um interpretador de linguagem: Este ponto passa por "forçar" uma interpretação sintáctica de regras JLups, "textuais", através de um canal comum e dedicado ao efeito. Para tal a fábrica de regras (ver secção 7.3.2.5) deverá ser totalmente reestruturada, respeitando a sua interface. Esta implementação poderá passar pela utilização de uma biblioteca dedicada (p.e. JavaCC[7]), ou poderá passar pela reutilização de um interpretador já existente;
- Reformulação do *core* das regras: Derivado do ponto anterior, a implementação de regras JLups deverá sofrer um reformulação para se adequar à nova sintaxe;
- Novas regras suportando novas funcionalidades: Posteriormente à introdução da nova gramática, deverão se implementadas novas operações úteis e básicas, como as comummente disponíveis na linguagem Prolog;
- Invocação de métodos Java: Por reflexão o JLups deverá conseguir invocar métodos de classes Java;
- Negação Explícita: Deverá ser analisada a possibilidade de introduzir a negação explícita no JLups. Este desenvolvimento poderá passar pela actualização e novo desenvolvimento sobre a biblioteca Mandarax;
- Variáveis *Grounded*: Poderá ser analisada a resolução do problema das variáveis *not grounded* ao nível da sintaxe JLups[8];
- Desenho de regras para arquitectura DBI: Estudo para codificação de regras para permitir a utilização de uma arquitectura BDI em JLups;
- *Java Rule Engine API*: Toda a evolução e desenvolvimento a ser realizado deve entrar em conformidade com a JSR 94[9][10], que define uma API simples para aceder a motores de regras com clientes Java SE e Java EE.

[6] Um agente foi numa fase de testes já integrado num ESB (BEA AquaLogic Service Bus) no entanto todos os agentes de serviços deverão ser integrados numa plataforma deste tipo.

[7] Ver http://javacc.dev.java.net/.

[8] O motor de prova interno do JLups actualmente pode eliminar o problema de *floundering*, sendo que este pesa sobre o mecanismo de prova (ver secções 7.3.1.6 e 7.3.1.7).

[9] Ver http://www.jcp.org/en/jsr/detail?id=094.

[10] Ver http://java.sun.com/developer/technicalArticles/J2SE/JavaRule.html.

O componente AgentServices, dada a sua "genericidade" também apresenta um conjunto de evoluções previstas, que aumentam o seu potencial e escopo:

- Cisão: Este componente poderá ser separado em sub-componentes, nomeadamente um para o nível de *messaging* entre agentes e outro para controlo de plataformas de agentes;
- *Web Service*: Expor a interface de comunicação com agentes via *web service*, de forma a elevar este componente a serviço genérico e facilitar a sua introdução numa SOA;
- *Messaging*: Melhorar as funcionalidades da camada *proxy* para agentes, de forma a aumentar os tipos de mensagens que circulam no canal.

Todas estas evoluções nestes dois componentes demonstram a utilidade da sua identificação e desenho. Após terminar o desenvolvimento inicial desta plataforma, é possível avançar e aprofundar a utilização destes componentes não alterando as suas interfaces de forma significativa, sendo possível prever a sua reutilização para outros fins. Da mesma forma a plataforma Susy deverá sofrer alguma actualizações dada esta "prova de conceito" bem sucedida.

Glossário

A

ACL Agent Communication Language, pág. 21.

AJAX Asynchronous Javascript And XML, pág. 2.

B

B2B Business-to-Business, pág. 39.

BAM Business Activity Monitoring, pág. 41.

BDI Belief, Desires and Intentions, pág. 24.

C

CA Controller Agent, pág. 59.

CORBA Common Object Request Broker Architecture, pág. 35.

CVS Concurrent Version System, pág. 113.

D

DAI Distributed Artificial Intelligence, pág. 8.

DCOM Distributed Component Object Model, pág. 35.

DLP Dynamic Logic Programming, pág. 31.

dMARS distributed MultiAgent Reasoning System, pág. 25.

DPS Distributed Problem Solving, pág. 8.

DSA Document Summarization Agent, pág. 60.

DWR Direct Web Remoting, pág. 69.

E

EAI Enterprise Application Integration, pág. 39.

EJB Enterprise Java Bean, pág. 67.

ESB Enterprise Service Bus, pág. 39.

F

FIPA Foundation for Intelligent Physical Agents, pág. 23.

G

GA Grammatical Agent, pág. 60.

GC Garbage Collector, pág. 10.

GUI Graphical User Interface, pág. 69.

H

HTML HyperText Markup Language, pág. 153.

I

IA Inteligência Artificial, pág. 1.

IDL Interface Definition Language, pág. 35.

IR Information Retrieval, pág. 15.

J

JAAS Java Authentication and Authorization Service, pág. 71.

JACK The Java Agent Kernel, pág. 27.

JADE Java Agent DEvelopment Framework, pág. 56.

JAM The Java Agent Model, pág. 27.

Java EE Java Platform, Enterprise Edition, pág. 55.

Java SE Java Platform, Standard Edition, pág. 56.

JAX-WS Java API for XML Web Services, pág. 57.

JCA J2EE Connector Architecture, pág. 38.

JDBC Java Database Connectivity, pág. 71.

JLups Java Language of dymamic UPdateS, pág. 57.

JMS Java Message Service, pág. 41.

JNI Java Native Interface, pág. 57.

JSP Java Server Page, pág. 56.

JSTL JavaServer Pages Standard Tag Library, pág. 70.

K

KQML Knowledge Query Manipulation Language, pág. 23.

L

LDAP Lightweight Directory Access Protocol, pág. 71.

LUPS Language of dymamic UPdateS, pág. 33.

M

MAS Multi-Agent Systems, pág. 21.

MUD Multi User Dimension, pág. 11.

MVC Model-View-Controller, pág. 66.

N

NA Nanny Agent, pág. 59.

NAF Negation As Failure, pág. xi.

P

PA Proxy Agent, pág. 59.

PAI Parallel AI, pág. 8.

PAM Pluggable Authentication Module, pág. 71.

PDA Personal Digital assistant, pág. 19.

PRS Procedural Reasoning System, pág. 27.

R

REST Representational State Transfer, pág. 36.

RuleML Rule Markup Language, pág. 158.

RUP Rational Unified Process, pág. 2.

S

SA Search Agent, pág. 60.

SOA Service Oriented Architecture, pág. 35.

SOAP Simple Object Access Protocol, pág. 36.

SQL Structured Query Language, pág. 70.

T

TCP Transmission Control Protocol, pág. 24.

TF-ISF Term-Frequency Inverse-Sentence-Frequency, pág. 91.

TI Tecnologias de Informação, pág. 21.

U

UA User Agent, pág. 59.

UDDI Universal Description, Discovery and Integration, pág. 36.

UML Unified Modeling Language, pág. 2.

URL Uniform Resource Locator, pág. 56.

W

WSDL Web Services Description Language, pág. 36.

WWW World Wide Web, pág. 7.

X

XML eXtensible Markup Language, pág. 41.

Bibliografia

[ALP+98] J. J. Alferes, J. Leite, L. M. Pereira, H. Przymusinska, and T. Przymuzinski. Dynamic logic programming. In *Proc. of KR'98*, 1998.

[AP02] J. Alferes and L. Pereira. Logic programming updating: a guided approach, 2002.

[APP+99a] J. J. Alferes, L. M. Pereira, H. Przymusinska, T. C. Przymusinski, and P. Quaresma. Preliminary exploration on actions as updates. In M. C. Meo and M. Vilares-Ferro, editors, *Procs. of the 1999 Joint Conference on Declarative Programming (AGP'99)*, pages 259–271, L'Aquila, Italy, September 1999.

[APP99b] José Júlio Alferes, Luís Moniz Pereira Halina Przymusinska, and Teodor C. Przymusinski. Lups - a language for updating logic programs. *In LPNMR'99, Lectures Notes in AI 1730*, pages 162–176, 1999.

[APPP99] José Júlio Alferes, Luis Moniz Pereira, Halina Przymusinska, and Teodor C. Przymusinski. LUPS - a language for updating logic programs. In *Logic Programming and Non-monotonic Reasoning*, pages 162–176, 1999.

[APPQa] José Júlio Alferes, Luís Moniz Pereira Halina Przymusinska, Teodor C. Przymusinski, and Paulo Quaresma. Dynamic knowledge representation and its applications.

[APPQb] José Júlio Alferes, Luís Moniz Pereira Halina Przymusinska, Teodor C. Przymusinski, and Paulo Quaresma. Preliminary exploration on actions as updates.

[AR96] P. Agre and S. Rosenschein. *Computational Theories of Interaction and Agency.* MIT Press, 1996.

[AR99] L. Andersson and Å. Rönnbom. Intelligent agents - a new technology for future distributed sensor systems? Master's thesis, School of Economies and Commercial Law Göteborg University, 1999.

[Bra87] M. E. Bratman. *Intention, Plans, and Practical Reason.* Center for the Study of Language and Inf, 1987.

[Bra97] Jeffrey M. Bradshaw. *Software Agents.* MIT Press, 1997.

[BRHL99] P. Busetta, R. Ronnquist, A. Hodgson, and A. Lucas. Jack intelligent agents - components for intelligent agents in java, 1999.

[BYRN99] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison Wesley Longman, 1999.

[CAC04] David Camacho, Ricardo Aler, and Juan Cuadrado. Rule-Based Parsing for Web Data Extraction . In Masoud Mohammadian, editor, *Intelligent Agents for Data Mining and Information Retrieval* , pages 64–86. Idea Publishing Group, ISBN : 1-59140-277-8, 2004.

[Cha04] David A. Chappell. *Enterprise service bus.* 2004.

[DSW97] K. Decker, K. Sycara, and M. Williamson. Middle-agents for the internet. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 1997.

[Ehl01] P. Ehlert. Intelligent driving agents. Master's thesis, Delft University of Technology, 2001.

[FG96] S. Franklin and A. Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In *Intelligent Agents III. Agent Theories, Architectures and Languages (ATAL'96)*, volume 1193, Berlin, Germany, 1996. Springer-Verlag.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, 2000. Chair-Richard N. Taylor.

[Fon97] Leonard N. Foner. Entertaining agents: A sociological case study. In *First International Conference on Autonomous Agents*, Marina Beach Marriott Hotel, Marina del Rey, California, February 1997. ACM, MIT Media Labs.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[GL87] M. P. Georgeff and A. L. Lansky. Procedural knowledge. In *IEEE Special Issue on Knowledge Representation*, volume 74, pages 1383–1398, 1987.

[GPP+99] Mike Georgeff, Barney Pell, Martha Pollack, Milind Tambe, and Mike Wooldridge. The belief-desire-intention model of agency. In Jörg Müller, Munindar P. Singh, and Anand S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555, pages 1–10. Springer-Verlag: Heidelberg, Germany, 1999.

[Gra98] Mark Grand. *Patterns in Java - A Catalog of Reusable Design Patterns Illustrated with UML*, volume 1. Wiley Computer Publishing, 1998.

[HR95] Barbara Hayes-Roth. An architecture for adaptive intelligent systems. *Special volume on computational research on interaction and agency, part 1*, 72(1):329–365, January 1995.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process.* Addison Wesley Longman, 1999.

[JW98] N. R. Jennings and M. J. Wooldridge. Applications of intelligent agents. *Agent Technology: Foundations, Applications and Markets*, pages 3–28, 1998.

[KG91] David Kinny and Michael P. Georgeff. Commitment and effectiveness of situated agents. In *Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 82–88, Sydney, Australia, 1991.

[KGR96] David Kinny, Michael Georgeff, and Anand Rao. A methodology and modelling technique for systems of BDI agents. In Rudy van Hoe, editor, *Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World*, Eindhoven, The Netherlands, 1996.

[KMA+01] Craig A. Knoblock, Steven Minton, Jose Luis Ambite, Naveen Ashish, Ion Muslea, Andrew Philpot, and Sheila Tejada. The ariadne approach to web-based information integration. *International Journal of Cooperative Information Systems*, 10(1-2):145–169, 2001.

[Les95] Victor R. Lesser. Multiagent systems: An emerging subdiscipline of ai. *ACM Computing Surveys*, 27(3):340–342, Sept 1995.

[Mae94] P. Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37(7):31–40, 1994.

[Mae95] P. Maes. Artificial life meets entertainment: Lifelike autonomous agents. *Communications of the ACM*, 38(11):108–114, 1995.

[Mar03] P. Marques. Component-based development of mobile agent systems. Master's thesis, Universidade de Coimbra, Jul 2003.

[MJ98] Richard Murch and Tony Johnson. *Intelligent Software Agents.* Prentice Hall PTR, 1st edition, 1998.

[Nwa96] Hyacinth S. Nwana. Software agents: An overview. *Knowledge Engineering Review*, 11(3):1–40, Sept 1996.

[Pal03] José Palmeiro. Sue - sumarizador extractivo java. Technical report, Universidade de Évora, Rua Romão Ramalho, 59, 7000-671 Évora, Jul 2003.

[Pet96] Charles J. Petrie. Agent-based engineering, the web, and intelligence. *IEEE Expert: Intelligent Systems and Their Applications*, 11(6):24–29, Decembre 1996.

[PQ05] José Palmeiro and Paulo Quaresma. Jlups, a java rule based engine for knowledge updates. Technical report, Universidade de Évora, Rua Romão Ramalho, 59, 7000-671 Évora, Apr 2005.

[PRN02] Thiago Pardo, Lucia Rino, and Maria Nunes. Extractive summarization: how to identify the gist of a text. *In the Proceedings of the International Information Technology Symposium - I2TS*, 2002.

[PRN03] Thiago Pardo, Lucia Rino, and Maria Nunes. Gistsumm: A summarization tool based on a new extractive method. *6th Workshop on Computational Processing of the Portuguese Language - Written and Spoken*, pages 210–218, 2003.

[RG91] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991.

[RG95] A. S. Rao and M. P. Georgeff. BDI-agents: from theory to practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.

[SM02] Stephen Stelting and Olav Maassen. *Applied Java Patterns*, volume 1. Sun Microsystem Press, 2002.

[SV97] P. Stone and M. Veloso. Multi-agent systems: A survey from a machine learning perspective. Technical report, Department of Computer Science,Carnegie Mellon University, 1997. Technical Report 193.

[Syc98] Katia Sycara. Multiagent systems. *AI Magazine*, 10(2):79–93, 1998.

[Vla03] Nikos Vlassis. A concise introduction to multiagent systems and distributed AI. Informatics Institute, University of Amsterdam, September 2003.

[Wei99] Gerhard Weiss, editor. *Multiagent systems : a modern approach to distributed artificial intelligence.* MIT Press, Cambridge, Mass., 1999.