UNIVERSIDADE DO VALE DO RIO DOS SINOS

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

PROGRAMA INTERDISCIPLINAR DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO APLICADA

Letícia Rafaela Rheinheimer

***WSAgent*:**

Um Agente Baseado em *Web Services* para Promover a Interoperabilidade entre Sistemas Heterogêneos no Domínio da Saúde

São Leopoldo

2004

Letícia Rafaela Rheinheimer

***WSAgent*:**

Um Agente Baseado em *Web Services* para Promover a Interoperabilidade entre Sistemas Heterogêneos no Domínio da Saúde

Monografia apresentada à Universidade do Vale do Rio dos Sinos como requisito parcial para a obtenção do título de Mestre em Computação Aplicada

Orientador: Prof. Dr. Sérgio Crespo C. S. Pinto

São Leopoldo

2004

Ficha catalográfica elaborada pela Biblioteca da
Universidade do Vale do Rio dos Sinos

R469w Rheinheimer, Letícia Rafaela

WSAgent: um agente baseado em Web Services para promover a interoperabilidade entre sistemas heterogêneos no domínio da saúde / por Letícia Rafaela Rheinheimer. - 2005.
148 f.

Dissertação (mestrado) - Universidade do Vale do Rio dos Sinos, Programa de Pós-Graduação em Computação Aplicada, 2005.

"Orientação: Profº. Drº. Sérgio Crespo C. S. Pinto, Ciências Exatas e Tecnológicas".

1. Desenvolvimento de sistemas. 2. Frameworks. 3. Web Services. 4. Base de dados heterogênea. 5. Design patterns. 6. Ontologia. I. Título.

CDU 004.414.2

Catalogação na Publicação:
Bibliotecária Eliete Mari Doncato Brasil – CRB10/1184

Meus agradecimentos...

...ao professor Sérgio Crespo, pela orientação e paciência durante os dois anos de mestrado.

...à empresa InfoSaúde Tecnologia de Informações Ltda, pelo apoio financeiro e informações necessárias ao desenvolvimento do trabalho.

...aos meus colegas, com quem foi possível dividir a caminhada e trocar experiências. Em especial, a Júnior Martins, que contribuiu para a construção do protótipo.

...à minha mãe Helena, à minha irmã Francesca e a meu namorado Natan, pelo apoio, o ombro amigo e o constante incentivo nas horas difíceis.

...aos meus amigos, em especial: Vinícius, Elisângela, Daniela e Genessa, pela ajuda, o companheirismo, o carinho e os momentos de conversa.

Vocês contribuíram muito para a minha caminhada até aqui.

RESUMO

Após o advento da *Internet*, diversas estratégias de desenvolvimento de *software* foram modificadas para promover maior reuso e interoperabilidade. *Design Patterns* e *Frameworks* nos ajudam a criar *software* e *design* flexíveis. A idéia de compor aplicações para que trabalhem juntas é bastante atrativa. No entanto, no domínio da saúde, surgem diversos empecilhos para que se realize esta integração. O uso de tecnologias de Agentes em conjunto com *Web Services* nos permite pensar em uma solução que garanta interoperabilidade, reuso e flexibilidade entre ambientes heterogêneos. Este trabalho descreve a arquitetura de um Agente de *Software*, chamado *WSAgent* (que consiste de uma instância de um *Framelet* para o sub-domínio paciente, no domínio da saúde) e suas estratégias de colaboração e interoperabilidade. Este trabalho também apresenta um estudo de caso com implementação de um protótipo.

Palavras-chave: *Frameworks*. *Design Patterns*. Ontologias. Agentes de *Software*. *Web Services*. Bases de Dados Heterogêneas.

ABSTRACT

After the Internet advent, several strategies about software development were changed to promote more reuse and interoperability. Design Patterns and Frameworks help us to create software and design flexible. The idea of glue applications to work together is very attractive. In the health domains, there are many drawbacks to address its goals. The use of agent technologies combined with Web Services allow us to think about the construction of a bind to grant interoperability, reuse and flexibility between heterogeneous environments. This work describes the architecture of a software agent called WSAgent – an instance of a Framelet of Patient subdomain in Health domain – and its strategies of collaborations and interoperability. This work also presents a case study with the implementation of a prototype.

Keywords: *Frameworks. Design Patterns. Ontologies. Software Agents. Web Services. Heterogeneous Databases.*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACL – Agent Communication Language

API – Application Programming Interface

B2B – Business to Business

CCOW – Clinical Context Object Workgroup

CDA – Clinical Document Architecture

CIAS – Clinical Image Access Service

COAS – Clinical Observation Access Service

COM – Component Object Model

CORBA – Common ORB Architecture

DAML – DARPA Agent Markup Language

DARPA – Defense Advanced Research Projects Agency

DCE – Distributed Computing Environment

DCOM – Distributed COM

DPRA – Document Patient Record Architecture

DSTU – Draft Standards for Trial Use

DTD – Document Type Definition

ebXML – eletronic business XML

EDI – Enterprise Data Interchange

EJB – Enterprise Java Bean

F-Logic – Frame Logic

FTP – File Transfer Protocol

GoF – the Gang of Four

HL7 – Health Level Seven

HMV – Hospital Moinhos de Vento

HP – Hewlett Packart

HTML – Hypertext Markup Language

HTTP – Hypertext Transfer Protocol

IBM – International Business Machines

IDL – Interface Definition Language

IIOP – Internet Inter-ORB Protocol

ISO – Internacional Organization for Standardization

JAXB – Java Architecture for XML Binding

JAXP – Java API for XML Processing

JAXR – Java API for XML Registries

JAX-RPC – Java API for XML-based RPC

JDBC – Java Database Connectivity

JSP – Java Server Pages

JSTL – JavaServer Pages Standard Tag Library

J2EE – Java 2 Enterprise Edition

JWSDP – Java Web Services Development Pack

KIF – Knowledge Interchange Format

KQML – Knowledge Query and Manipulation Language

LAN – Local Area Network

MDF – Message Development Framework

MIME – Multipurpose Internet Mail Extensions

NAICS – North American Industry Classification System

OIL – Ontology Inference Layer

OMA – Object Management Architecture

OMG – Object Management Group

OO – Orientado a Objetos

OOPSLA – Object-Oriented Programming, Systems, Languages, and Applications

ORB – Object Request Broker

OWL – Web Ontology Language

PDA – Personal Digital Assistant

PIS – Person Identification Service

QoS – Quality of Service

RDF – Resource Description Framework

RIM – Reference Information Model

RMI – Remote Method Invocation

RPC – Remote Procedure Call

SAAJ – SOAP with Attachments API for Java

SMTP – Simple Mail Transfer Protocol

SOAP – Simple Object Access Protocol

SOC – Service Oriented Computing

SQL – Structured Query Language

TCP – Transfer Control Protocol

UDDI – Universal Description, Discovery and Integration

UNIX – UNiplexed Information and Computing System

UNSPSC – Universal Standard Products and Services Classification

VAN – Value Added Network

WSDL – Web Services Description Language

W3C – World Wide Web Consortium

XML – eXtensible Markup Language

XMLITS – XML Implementation Technology Specification

SUMÁRIO

1 INTRODUÇÃO

As tecnologias de *Web Services* não surgiram como algo completamente novo. Elas têm base na evolução da arquitetura de componentes e seus conceitos se originam no paradigma da computação distribuída, que surgiu com o advento das redes de computadores (CAULDWELL, 2001; DURHAM, 2001; GUNZER, 2002; RIST, 2002; YANG, 2003).

No princípio, aplicações eram executadas de forma centralizada em *mainframes*. Mais tarde, terminais foram conectados a estes computadores, permitindo o trabalho a partir da digitação de comandos em formato texto. Passados mais alguns anos, surgiram os computadores pessoais e uma nova maneira de desenvolver e utilizar aplicações. Nesta época – anos 80 –, o desafio era fazer com que as aplicações "conversassem" umas com as outras na mesma máquina, não havendo muita preocupação com protocolos de comunicação (CAULDWELL, 2001; DURHAM, 2001).

O surgimento da computação distribuída fez com que as aplicações fossem divididas, constituindo uma arquitetura de duas camadas: cliente e servidor. Em um primeiro momento, esta arquitetura trouxe flexibilidade ao projeto de aplicações. Porém, a fim de melhorar o tratamento de falhas e a escalabilidade, uma terceira camada foi introduzida. A arquitetura resultante separou as aplicações em um nível de apresentação, um nível contendo as regras de negócio e um nível responsável pela manutenção dos dados, tornando-se a maneira mais popular de fazer aplicações. Nesta época, o conceito de serviço poderia ser descrito como a

capacidade de realizar comunicação entre as partes distribuídas da arquitetura. A base para esta comunicação era o protocolo *Remote Procedure Call*[1] (*RPC*) (GUNZER, 2002; RIST, 2002).

Essa divisão da arquitetura decorreu na criação de mais uma nova camada de *software* – chamada *middleware* –, a fim de poupar os desenvolvedores de detalhes de implementação de baixo nível para diferentes máquinas. Mas, naquele momento, para as empresas, o ponto chave era manter uma central de dados bem integrada. Suportar diferentes ambientes e plataformas não era tão importante, e áreas como a de *Enterprise Data Interchange*[2] (*EDI*) eram entidades à parte dentro da organização (LEVITT, 2001; GUNZER, 2002; OGBUJI, 2002a).

Após o insucesso de tecnologias de *middleware* como transferência de mensagens e *Distributed Computing Environment*[3] (*DCE*), começou a emergir – no início dos anos 90 – uma nova geração de computação distribuída. Os sistemas de informação tornaram-se parte fundamental da estratégia de empresas e a integração de sistemas tornou-se uma das mais importantes considerações na escolha de tecnologia. A orientação a objetos foi adotada no desenvolvimento de sistemas, devido às expectativas de reuso, maior facilidade de manutenção, redução de custos e maior retorno nos investimentos. Assim, camadas baseadas em um modelo de programação procedural foram suplantadas por camadas baseadas em programação orientada a objetos como *Component Object Model*[4] (*COM*) e *Common Object Request Broker Architecture*[5] (*CORBA*) (CAULDWELL, 2001; DURHAM, 2001; GUNZER, 2002; OGBUJI, 2002a).

Na década de 90, com uma maior difusão das *Local Area Networks* (LANs), a comunicação entre máquinas tornou-se prioridade. Camadas como *COM* e *CORBA* foram extendidas para suportar a comunicação através de redes, resultando nos protocolos

[1] Informações podem ser encontradas em: <http://www.faqs.org/rfcs/rfc1050.html>
[2] *EDI* surgiu, em 1975, como a primeira tentativa de criar uma forma de comunicação padrão para negócios, via rede de computadores (LEVITT, 2001)
[3] Informações podem ser encontradas em: <http://www.opengroup.org/dce/>
[4] Informações podem ser encontradas em: <http://www.microsoft.com/com/>
[5] Informações podem ser encontradas em: <http://www.omg.org/technology/documents/ corba_spec_catalog.htm>

Distributed COM[6] (*DCOM*) e *Internet Inter-ORB Protocol*[5] (*IIOP*). Durante este período, outras tecnologias mais especializadas foram surgindo, e a popularidade da linguagem *Java* levou ao desenvolvimento de seu protocolo *Remote Method Invocation*[7] (*RMI*) (CAULDWELL, 2001; DURHAM, 2001; LEVITT, 2001; GUNZER, 2002; OGBUJI, 2002a).

Os protocolos existentes, até então, não eram interoperáveis e apresentavam dificuldades em operar através de *firewalls* ou servidores *proxy*. Isto contribuiu para que desenvolvedores procurassem soluções que oferecessem o baixo acoplamento das tecnologias de transferência de mensagens e a onipresença da *Internet* (CAULDWELL, 2001; GUNZER, 2002).

O surgimento da *eXtensible Markup Language*[8] (*XML*), em 1990, veio permitir a representação de dados e a troca de mensagens entre sistemas de forma auto-descritiva, extensível e independente de plataforma. Uma das melhores implementações existentes de protocolo baseado em *XML* é o *XML-RPC*[9]. A *XML* tornou-se padrão em 1998 e tornou-se a escolha lógica para uma comunicação aplicação-para-aplicação padronizada, abrindo caminho para outras tecnologias de *Web Services* (CAULDWELL, 2001; LEVITT, 2001; COYLE, 2002).

Neste mesmo ano, uma especificação para troca estruturada de documentos *XML* começou a crescer dentro de um pequeno grupo de organizações que incluía a *Microsoft*: o protocolo *Simple Object Access Protocol* (*SOAP*), sucessor do protocolo *XML-RPC*, foi lançado em 2000. Dos esforços de empresas como *IBM*, *Microsoft* e *Ariba*, veio a *Web Services Description Language* (*WSDL*) e, mais tarde, o sistema de diretórios *Universal Description, Discovery and Integration* (*UDDI*) (DURHAM, 2001; LEVITT, 2001; OGBUJI, 2002a).

A *XML* e os protocolos da *Internet* revolucionaram o pensamento da indústria de *EDI*. Primeiro, *EDI* sobre *Simple Mail Transfer Protocol* (*SMTP*) e *Hypertext Transfer Protocol* (*HTTP*) surgiram como uma maneira de esquivar-se das caras taxas das *Value Added*

[6] Informações podem ser encontradas em: <http://www.microsoft.com/com/tech/dcom.asp>
[7] Informações podem ser encontradas em: <http://java.sun.com/products/jdk/rmi/>
[8] Informações podem ser encontradas em: <http://www.w3.org/xml/>
[9] Informações podem ser encontradas em: <http://www.xmlrpc.com/spec>

Networks (*VAN*s). Em seguida, grupos começaram a trabalhar em formas de codificar transações *EDI*. Isto levou às primeiras formas de *Web Services*, inspiradas não na integração de aplicações corporativas, mas em transações *Business to Bussiness* (*B2B*). O esforço de padronização destes primeiros sistemas *XML*/*EDI* resultou na *e-business XML*[10] (*ebXML*) (COYLE, 2002; OGBUJI, 2002a).

Em abril de 2001, ocorreu o *W3C Workshop on Web Services*, evento que tinha por objetivo planejar o futuro dos *Web Services*. O *World Wide Web Consortium*[11] (*W3C*), a partir de então, colocou as tecnologias de *Web Services* sob amparo de outros padrões como HTML[12] e *XML* (DURHAM, 2001; LEVITT, 2001; COYLE, 2002; OGBUJI, 2002a).

O ano 2002 foi de consolidação para os *Web Services*. Empresas como *HP*, *Microsoft*, *Sun Microsystems* e *IBM* vêm oferecendo plataformas de desenvolvimento e seguem formando grupos dentro do *W3C* para padronização de tecnologias para *Web Services*. *XML*, *SOAP*, *WSDL* e *UDDI* são as principais tecnologias utilizadas hoje (LEVITT, 2001; OGBUJI, 2002b; RIST, 2002).

Web Services constituem um paradigma emergente na área de computação distribuída e comércio eletrônico: o da computação orientada a serviços (*SOC*). Para operar em um ambiente *SOC*, as aplicações – serviços – devem definir seus requisitos e capacidades funcionais e não funcionais em um formato padronizado. Baseado em descrições de serviço declarativas e descoberta de serviços automatizada, seleção e ligação dinâmica tornam-se uma capacidade nativa de aplicações e *middleware SOC*. Nesta arquitetura componentizada, serviços tornam-se os blocos de construção básicos, a partir dos quais novas aplicações são criadas (CRESPO, 2000; RIST, 2002; CURBERA, 2003; PAPAZOGLOU, 2003; YANG, 2003).

[10] Informações podem ser encontradas em: <http://www.ebxml.org>
[11] O *site* oficial do *W3C* pode ser acessado em: <http://www.w3.org/>
[12] Informações podem ser encontradas em: <http://www.w3.org/MarkUp/>

Na área de negócios, *Web Services* facilitam a integração de aplicações permitindo, a curto prazo, interligar novas aplicações com aplicações existentes e, a médio prazo, dar origem a um novo conceito de aplicações, proporcionando interoperabilidade a baixo custo (COYLE, 2002; FOSTER, 2003; GEER, 2003; LANGDON, 2003).

1.1 MOTIVAÇÃO E CONTEXTO DO TRABALHO

Apesar de emergentes, as tecnologias de *Web Services* vêm sendo cada vez mais utilizadas em âmbito corporativo. Uma de suas principais vantagens reside no fato de possibilitarem a integração de sistemas legados, com independência de plataforma, proporcionando agilidade e flexibilidade para este tipo de solução. A necessidade de integrar sistemas tem sido uma realidade para diversas empresas. (FOSTER, 2003; GEER, 2003; LANGDON, 2003).

Mas as tecnologias de *Web Services* oferecem outras possibilidades além da integração de sistemas legados. Sistemas inteiros podem ser desenvolvidos como um conjunto de serviços, podendo ser utilizados dentro da organização ou disponibilizados na *Internet*. Estes serviços podem ser utilizados de forma isolada, ou combinados para constituírem outros serviços. Ainda, processos de negócio já implementados por aplicações existentes podem ser disponibilizados na forma de serviços, tanto em uma *Intranet* como na *Internet*, sendo consumidos por outros sistemas e serviços. Isto contribui para o desenvolvimento de aplicações de comércio eletrônico e *B2B* (LANGDON, 2003; PAPAZOGLOU, 2003; TURNER, 2003).

Web Services constituem um novo paradigma de desenvolvimento. O surgimento da *Internet* trouxe a necessidade de novas estratégias de desenvolvimento e adaptação de outras já existentes. Isto tem levado a uma busca, cada vez maior, por atender a requisitos como qualidade, reuso e interoperabilidade. O uso das tecnologias de *Web Services* contribui para a

satisfação destes requisitos (COYLE, 2002; RAMBHIA, 2002; KOTOK, 2003; LANGDON, 2003; TURNER, 2003).

Isto pode ser maximizado através da utilização de *Frameworks* e *Design Patterns*, que nos permitem construir *software* flexível (GAMMA, 1995; FAYAD, 1999a, 1999c).

Ainda, técnicas provindas de outras áreas de pesquisa podem ser aplicadas. Exemplos podem ser encontrados em trabalhos de Tsai (2003), Van Den Heuvel (2003) e Limthanmaphon (2003), onde Ontologias, Agentes e outros recursos de Inteligência Artificial são utilizados na criação de serviços.

A área da saúde mostra-se um campo bastante rico para a aplicação das tecnologias mencionadas, tanto pela diversidade de sistemas legados e em desenvolvimento que é possível encontrar como pela crescente demanda na administração de dados clínicos (BIRD, 2000, 2002; KIM, 2001; KUHN, 2001).

Este trabalho está focado na integração de bases de dados heterogêneas, no domínio da saúde. A solução proposta será aplicada em um estudo de caso, resultando em uma implementação real. As empresas que fazem parte do estudo são a InfoSaúde Tecnologia de Informações Ltda e o Hospital Moinhos de Vento (HMV), ambas sediadas na cidade de Porto Alegre.

A InfoSaúde vem realizando um projeto para o HMV, que consiste no desenvolvimento de um sistema chamado Clínico Assistencial. Este sistema deve permitir o controle de tudo o que é realizado dentro do hospital, desde tarefas da equipe de limpeza até procedimentos realizados pelos médicos, e depende de informações provenientes do sistema corporativo do HMV para funcionar (entrada de pacientes no hospital, por exemplo). É preciso integrar o sistema Clínico Assistencial e o sistema legado, de forma que ambos possam funcionar conjuntamente.

1.2 PROBLEMA

O ponto crítico no desenvolvimento do trabalho está relacionado à parte de persistência da solução, que será responsável pela manipulação e armazenamento dos dados.

O maior problema encontrado é a obtenção de dados mais recentemente manipulados, que devem ser utilizados para sincronização das informações nas diferentes bases de dados envolvidas. A questão é: como identificar os registros que foram inseridos, alterados ou excluídos, para então refletir estas modificações na base que se deseja atualizar? É necessário desenvolver uma solução que possa ser aplicada a diversos tipos de bancos de dados.

Um outro problema, neste contexto, se refere ao formato no qual os dados são armazenados nas bases de dados. Entre informações equivalentes, pode haver diferenças quanto a tipos de dados, tamanho máximo permitido para os campos de tabela, dados que devem ser “quebrados” ou concatenados, e assim por diante. É preciso definir como a compatibilização das informações pode ser realizada.

A solução desenvolvida não pode comprometer a performance dos sistemas envolvidos.

1.3 QUESTÃO DE PESQUISA

A questão central deste trabalho é: como realizar a integração de plataformas heterogêneas, viabilizando a interoperabilidade entre elas, de forma que a solução desenvolvida proporcione reuso, qualidade e até mesmo competitividade para a empresa?

1.4 OBJETIVOS

Este trabalho tem como objetivo maior a aplicação das tecnologias de *Web Services* no desenvolvimento de Agentes de S*oftware*, que serão responsáveis pela integração de bases de dados heterogêneas.

A solução proposta pode ser melhor explicada tomando-se como exemplo o caso InfoSaúde. O sistema Clínico Assistencial realiza diversos processos, os quais desempenham algumas funções consideradas críticas – que precisam das informações extremamente atualizadas. Um destes processos, escolhidos para um estudo inicial, é aquele que realiza operações relacionadas a pacientes e leitos.

A fim de dar suporte às operações críticas sem sobrecarregar a rede do hospital com um fluxo constante de troca de dados, optou-se por uma solução que realize o processo de atualização apenas no momento em que uma operação crítica seja realizada. Para isto, cada processo terá um conjunto de Agentes associados às operações em questão.

FIGURA 1.1 – Agentes reagindo à execução de uma operação crítica.

A Fig. 1.1 exemplifica como deverá funcionar esta integração através do uso de *Web Services* no papel de Agentes. Quando um usuário do sistema Clínico Assistencial solicitar a execução de uma operação crítica, esta operação invocará a execução de um Agente que deverá verificar a base de dados do sistema. A seguir, este Agente invocará a execução de

outro, associado ao sistema legado, a fim de saber se há alguma informação mais recente. O Agente legado verificará a base de dados do sistema e retornará a resposta ao Agente Clínico Assistencial. Havendo dados mais recentemente modificados, as informações serão enviadas para o Agente Clínico Assistencial, a fim de serem tratadas e transferidas para a base de dados correspondente. Da mesma forma, dados gerados pelo sistema Clínico Assistencial e que sejam relevantes para o sistema legado, devem ser devidamente atualizados.

Note-se que um processo de conversão e compatibilização dos dados deve ser realizado sempre que um Agente legado receber informações de um Agente Clínico Assistencial e sempre que um Agente Clínico Assistencial receber dados para atualizar a base do sistema a que está associado.

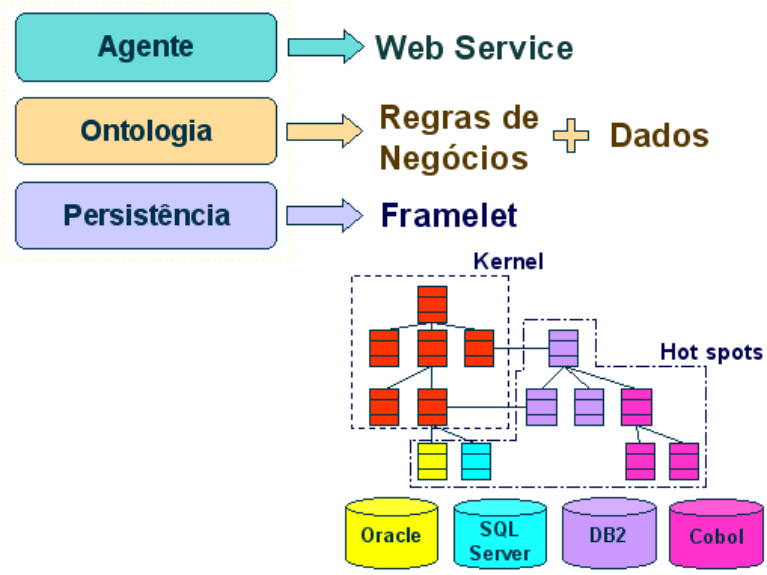


FIGURA 1.2 – Arquitetura da solução proposta.

A solução proposta possui uma arquitetura única – mostrada na Fig. 1.2 – que, para ser desenvolvida, exige a concretização de alguns objetivos específicos:

- especificar e implementar o Agente de *Software*;
- definir uma Ontologia de interoperabilidade, que será utilizada pelo Agente;

- especificar e implementar um *Framelet* de persistência, que deverá suportar os principais bancos de dados e permitir sua estensão para outros bancos que se deseje acrescentar à solução;
- definir as regras de negócio do Agente, para compatibilização com a camada de persistência.

Quando concluída a solução para o processo paciente-leito, devem ser feitas simulações para outras áreas do sistema Clínico Assistencial, a fim de validar a aplicação desenvolvida e efetuar alterações que venham a ser necessárias.

Os principais aspectos favoráveis identificados nesta solução são descritos a seguir:

- o uso de tecnologias de *Web Services*, baseadas em *XML*, proporciona interoperabilidade;
- com base na solução definida para o processo paciente-leito, o desenvolvimento para um novo processo exige apenas a definição de uma camada de Ontologia adequada. As demais camadas são reutilizadas, pois o funcionamento é o mesmo para todos os processos envolvidos;
- os Agentes gerados são independentes do código dos sistemas que devem integrar. Esta característica permite que a solução possa ser facilmente utilizada em sistemas de outros domínios, não havendo necessidade de reescrita de código para adaptação da solução;
- após os Agentes serem gerados e estarem operando, caso venha a ocorrer o acréscimo de um campo em uma tabela da base legada, por exemplo, apenas será necessário atualizar a Ontologia do Agente afetado por esta mudança. Isto fará com que o Agente saiba como tratar esta modificação, sem a necessidade de criar *triggers* (ou outros métodos auxiliares equivalentes) para adaptar o funcionamento da solução às modificações que venham a ocorrer nas bases de dados envolvidas.

Se a modificação for de maior porte, como a troca de uma base de dados *SQL Server* para uma base *Oracle*, por exemplo, será necessário regerar os Agentes para trabalharem com a nova base e adaptar as Ontologias correspondentes;

- os Agentes não precisam, necessariamente, estar localizados na mesma máquina ou servidor de aplicações. Isto porque *Web Services* permitem total distribuição: apenas é necessário saber o endereço através do qual podem ser invocados e quais parâmetros devem ser passados. Isto, como será visto em uma seção sobre *Web Services* contida nesta proposta, pode ser definido em tempo de compilação ou descoberto em tempo de execução, não havendo empecilhos quanto a esse aspecto para o funcionamento da solução.

Na região de atuação da empresa InfoSaúde não há concorrentes que utilizem uma solução semelhante. Assim, o trabalho pode ser incrementado pela empresa e vir a se tornar um diferencial para seus produtos, o que vem de encontro à questão da competitividade.

Cabe ressaltar que o desenvolvimento de mecanismos de segurança está fora do escopo deste trabalho. A relevância deste aspecto é reconhecida, mas este ponto não pode ser desenvolvido devido à necessidade de definir limites para a execução do trabalho. A questão da segurança está prevista para ser desenvolvida em trabalhos futuros.

1.5 ORGANIZAÇÃO DO TRABALHO

Este volume está divido em 6 capítulos, sendo o primeiro esta introdução. Os demais capítulos são descritos a seguir:

- capítulo 2: Revisão Bibliográfica – descreve as tecnologias utilizadas no desenvolvimento do trabalho. O texto fala sobre *Design Patterns*, *Frameworks*, Ontologias, Agentes de *Software* e *Web Services*;

- capítulo 3: Trabalhos Relacionados – mostra alguns trabalhos na área da saúde, com objetivos semelhantes, e que envolvem o uso de tecnologias descritas no capítulo anterior;
- capítulo 4: *WSAgent* – descreve a solução a ser desenvolvida e aplicada no estudo de caso;
- capítulo 5: Estudo de Caso – mostra a utilização da solução proposta na integração do processo paciente-leito, no contexto da empresa InfoSaúde;
- capítulo 6: Conclusão – apresenta algumas considerações quanto ao desenvolvimento do trabalho, indicando pontos de melhoria e a continuidade do desenvolvimento junto à empresa InfoSaúde.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo oferece uma visão geral das principais tecnologias aplicadas no trabalho: *Design Patterns*, *Frameworks*, Ontologias, Agentes de *Software* e *Web Services*.

2.1 *DESIGN PATTERNS*

O conceito de *Design Patterns* foi definido por Christopher Alexander, que o aplicou na área da arquitetura: “cada padrão descreve um problema que ocorre diversas vezes em nosso ambiente e descreve o coração da solução deste problema, de forma que você pode utilizar esta solução um milhão de vezes sem nunca fazê-lo duas vezes da mesma maneira” (ALEXANDER, 1977 *apud* GAMMA, 1995, p. 2).

Quando se trata de computação, este conceito também pode ser muito bem aplicado. Nesse contexto, *Design Patterns* constituem soluções para problemas recorrentes em Engenharia de *Software*. Cada padrão identifica classes e instâncias participantes com seus papéis, colaborações e a distribuição de responsabilidades, sendo que estes elementos podem ser customizados para resolver um problema num contexto particular. Assim, o uso de padrões deve contribuir para reutilização e flexibilização no desenvolvimento de *software* orientado a objetos (PREE, 1994, 1995, 1997; GAMMA, 1995; FAYAD, 1999b; CRESPO, 2000). A Fig. 2.1 traz um exemplo de padrão.

STATE Comportamental de objetos

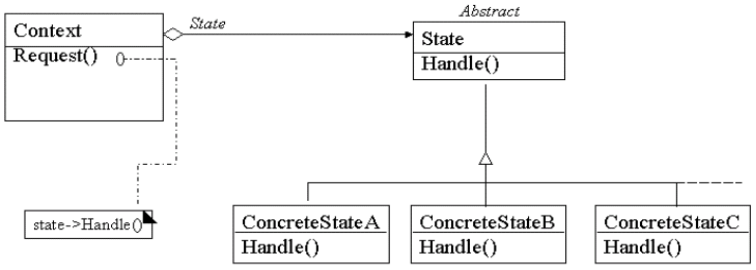
Intenção
Permite a um objeto alterar seu comportamento quando seu estado interno muda. O objeto parecerá ter mudado de classe.

Também conhecido como
Object for States

Aplicabilidade
Use o padrão *State* em um dos casos seguintes:

- O comportamento de um objeto depende do seu estado e ele pode mudar seu comportamento em tempo de execução, dependendo desse estado;
- Operações têm comandos condicionais grandes, de várias alternativas, que dependem do estado do objeto. Este estado é normalmente representado por uma ou mais constantes enumeradas. Freqüentemente, várias operações conterão esta mesma estrutura condicional. O padrão *State* coloca cada ramo do comando adicional em uma classe separada. Isto lhe permite tratar o estado do objeto como um objeto propriamente dito, que pode variar independentemente de outros objetos.

Estrutura



Participantes

- *Context*
  - Define a interface de interesse para os clientes.
  - Mantém uma instância de uma subclasse *ConcreteState* que define o estado corrente.
- *State*
  - Define uma interface para encapsulamento associado com um determinado estado ao *Context*.
- Subclasses *ConcreteState*
  - Cada subclasse implementa um comportamento associado com um estado do *Context*.

Padrões relacionados
O padrão *Flyweight* explica quando e como objetos *State* podem ser compartilhados. Objetos *State* são freqüentemente *Singletons*.

FIGURA 2.1 – Exemplo de descrição: *Design Pattern State* (RHEINHEIMER, 2002, p. 22).

Padrões de projeto são muito bem documentados – descritos de forma estruturada e catalogados –, contendo quatro elementos essenciais (GAMMA, 1995):

- nome do padrão;
- problema – descrição do contexto no qual pode-se utilizar o padrão, podendo ser a descrição de problemas de projeto específicos, ou de estruturas de classe/objeto, ou ainda uma lista de condições que devam ser satisfeitas para que faça sentido utilizá-lo;

- solução – descrição abstrata de um problema de projeto e como um arranjo geral de elementos (classes e objetos) resolve o mesmo;
- conseqüências – resultados e análises das vantagens e desvantagens de sua aplicação.

O exemplo da Fig. 2.1 descreve alguns itens do gabarito citado anteriormente. Existem diversos padrões disponíveis para consulta em livros e na *Internet*. Algumas fontes para consulta são: Gamma (1995), Duell (1997), Alur (2002), Marinescu (2002) e Hillside.Net (2004).

O padrão *State*, mostrado no exemplo anterior, pode ser compreendido a partir da visão trazida por Duell (1997), que não é vinculado ao contexto de *software*. Segundo esta visão, o padrão é explicado a partir do funcionamento de uma máquina de lanches, mostrada na Fig. 2.2.

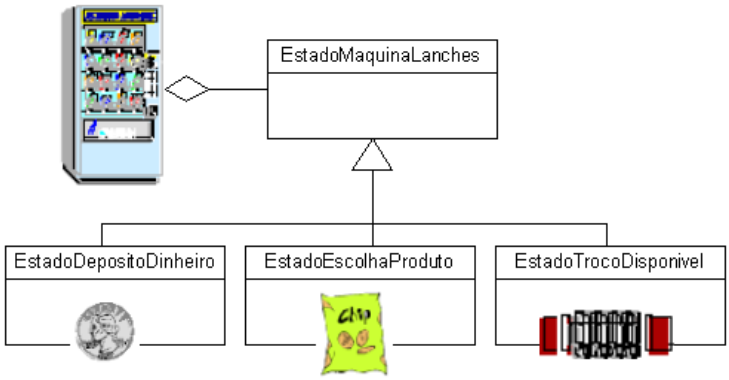


FIGURA 2.2 – Visão "não *software*" do *Design Pattern State*, adaptado de Duell (1997).

O comportamento de uma máquina de lanches varia com base em seu estado. O depósito de dinheiro, a seleção de um produto e o fornecimento de troco fazem parte do conjunto de estados da máquina. O padrão *State* permite que um objeto mude seu comportamento quando seu estado interno muda; assim, a máquina de lanches realizará uma determinada ação (ou conjunto de ações) de acordo com o estado em que se encontra.

Segue uma comparação entre as estruturas apresentadas na Fig. 2.1 e na Fig. 2.2:

- a classe `Context` da Fig. 2.1 corresponde à máquina de lanches. O depósito de dinheiro, o estoque de produtos e o dinheiro para troco formam o contexto da

solução. A máquina possui controle de seu estado atual, agindo em reação às ações executadas pelo cliente (a pessoa que busca adquirir um lanche na máquina);

- a classe `State` da Fig. 2.1 corresponde ao estado atual da máquina, representado por `EstadoMaquinaLanches` na Fig. 2.2. O estado atual sempre estará vinculado a um dos tipos de estado existentes para a máquina;
- as classes `ConcreteStateA`, `ConcreteStateB` e `ConcreteStateC` da Fig. 2.1 (subclasses de `State`) correspondem aos tipos de estado que a máquina pode apresentar. Na Fig. 2.2, os tipos de estado são `EstadoDepositoDinheiro`, `EstadoEscolhaProduto` e `EstadoTrocoDisponível`. O comportamento associado a cada componente do contexto corresponde a um estado (por exemplo, o depósito de dinheiro é representado por `EstadoDepositoDinheiro`).

Durante o funcionamento da máquina, o comportamento específico para um certo estado é localizado. Quando é fornecido um pacote de batatas fritas, a luz que indica que há troco a receber não irá acender, a menos que o troco seja requerido. As transições de estado são explícitas. O pacote de batatas fritas não será entregue, a menos que o dinheiro depositado seja suficiente para pagar pelo produto (DUELL, 1997).

2.1.1 Classificação de Padrões de Projeto

Os padrões definidos pelo grupo denominado *Gang of Four* (*GoF*) são considerados como base para os demais *Design Patterns*. De acordo com Gamma (1995), esses padrões podem ser classificados segundo as seguintes categorias:

- padrões de criação – abstraem o processo de instanciação. Ajudam a tornar o sistema independente de como os objetos são criados, compostos e representados. Um padrão de criação de classes usa a herança para variar a classe que é instanciada, enquanto

um padrão de criação de objetos delega a instanciação para outro objeto. Os padrões de criação dão muita flexibilidade quanto ao que é criado, quem cria, como e quando é criado. Permitem configurar um sistema com objetos “produto” que variam amplamente em estrutura e funcionalidade, sendo que esta configuração pode ser estática (especificada em tempo de compilação) ou dinâmica (especificada em tempo de execução);

- padrões estruturais – preocupam-se com a forma como classes e objetos são compostos para formar estruturas maiores. Os padrões estruturais de classes utilizam a herança para compor interfaces ou implementações. Este tipo de padrão é particularmente útil para fazer bibliotecas de classes desenvolvidas independentemente trabalharem juntas. Já os padrões estruturais de objetos, em lugar de compor interfaces ou implementações, descrevem maneiras de compor objetos para obter novas funcionalidades. A flexibilidade provém da capacidade de mudar a composição em tempo de execução, o que é impossível com a composição estática de classes;
- padrões comportamentais – preocupam-se com algoritmos e a atribuição de responsabilidades entre objetos. Não descrevem apenas padrões de objetos ou classes, mas também os padrões de comunicação entre eles. Estes padrões caracterizam fluxos de controle difíceis de seguir em tempo de execução; eles afastam o foco do fluxo de controle, para que seja possível concentrar-se somente na maneira como os objetos são interconectados. Padrões comportamentais de classes utilizam herança para distribuir o comportamento entre classes, enquanto padrões comportamentais de objetos utilizam a composição de objetos. Alguns descrevem como um grupo de objetos pares (*peer objects*) cooperam para a execução de uma tarefa que nenhum objeto sozinho poderia executar por si mesmo.

2.1.2 Utilização de Padrões

Decompor um sistema em objetos não é uma tarefa fácil: eles nem sempre são encontrados na fase de análise e nos estágios iniciais de projeto. Existem diversos fatores que contribuem para dificultar esse processo: encapsulamento, granularidade, dependência, flexibilidade, desempenho, evolução e reutilização, entre outros. Como vencer estes desafios? *Design Patterns* podem ajudar (GAMMA, 1995; APPLETON, 1998).

Design Patterns facilitam a identificação de abstrações menos óbvias, bem como os objetos que podem capturá-las. Por exemplo: como decidir o que deve ser um objeto? Existem padrões que descrevem maneiras de representar subsistemas completos como objetos – *Design Pattern Facade* –, como suportar enormes quantidades de objetos nos níveis de granularidade mais finos – *Design Pattern Flyweight* –, como decompor objetos em objetos menores – *Design Patterns Builder* e *Command* (GAMMA, 1995; PORTLAND PATTERN REPOSITORY, 2004).

Também existem padrões que ajudam a definir interfaces de objetos, através da identificação de seus elementos-chave e tipos de dados que são enviados através da interface. Um exemplo é o *Design Pattern Memento*. Há ainda outros padrões que especificam relacionamentos entre as interfaces. Tudo isso é muito importante, pois em um sistema Orientado a Objetos (OO) só é possível conhecer um objeto, solicitar-lhe algo, através de sua interface (GAMMA, 1995; PORTLAND PATTERN REPOSITORY, 2004).

Padrões de criação, como *Abstract Factory*, *Builder* e *Singleton*, permitem abstrair o processo de criação de objetos, propiciando várias maneiras de associar uma interface com sua implementação de forma transparente no momento da instanciação. Isto reduz dependências de implementação entre subsistemas, favorecendo a reutilização. Em outras palavras, se algum aspecto da implementação herdada de uma classe não for apropriada para novos domínios de

problemas, a classe mãe deve ser reescrita ou substituída por algo mais apropriado. Esta dependência limita a flexibilidade, o que pode ser resolvido com herança de classes abstratas, que fornecem pouca ou nenhuma implementação – o que é comum aos *Design Patterns* (GAMMA, 1995; PORTLAND PATTERN REPOSITORY, 2004).

Segundo Gamma (1995), *Design Patterns* não devem ser usados indiscriminadamente. Devem apenas ser aplicados quando a flexibilidade que oferecem é realmente necessária, sob pena de haver perdas em relação a custo/benefício de sua utilização; ou seja, deve-se avaliar a seção que fala a respeito de conseqüências, na descrição dos padrões.

2.2 *FRAMEWORKS*

Segundo alguns autores, um *Framework* pode ser considerado:

- "(...) um conjunto de objetos que colaboram com o objetivo de cumprir um conjunto de responsabilidades para uma aplicação específica ou um domínio de aplicação" (JOHNSON, 1991 *apud* CRESPO, 2000, p. 7);
- "(...) uma arquitetura desenvolvida com o objetivo de se obter a máxima reutilização, representada como um conjunto de classes abstratas e concretas, com grande potencial de especialização" (MATTSON, 1996 *apud* CRESPO, 2000, p. 7);
- "(...) um conjunto de classes que constitui um *design* abstrato para soluções de uma família de problemas" (JOHNSON, 1998 *apud* CRESPO, 2000, p. 7).

Segundo Pree (1999a), *Frameworks* unificam sistemas de *software* para um domínio específico e constituem uma construção semi-acabada de blocos de códigos prontos para uso, em associação com uma arquitetura global, obtida pela composição e interação entre os blocos. Os aspectos de um *Framework* que não são projetados para adaptação – chamados *frozen spots* – são representados por meio de métodos *template* e constituem o *kernel* do *Framework*.

Há também os aspectos que são projetados para adaptação, chamados *hot spots*. Crespo (2000, p. 13) diz que "um *hot spot* é uma classe abstrata que não possui implementação e deve ser especializada para necessidades específicas da aplicação a ser gerada". Diz também que "a especialização pode ser realizada tanto por herança como por delegação, dependendo da forma como os *hot spots* foram planejados".

Hot spots são representados por métodos *hook*. Cada *hook* tem como foco um pequeno requisito. Desta forma, para problemas complexos serão necessários vários *hooks*. Cada *hook* detalha as mudanças no *design* e os efeitos que este terá sobre o *Framework* (CRESPO, 1998; FAYAD, 1999a).

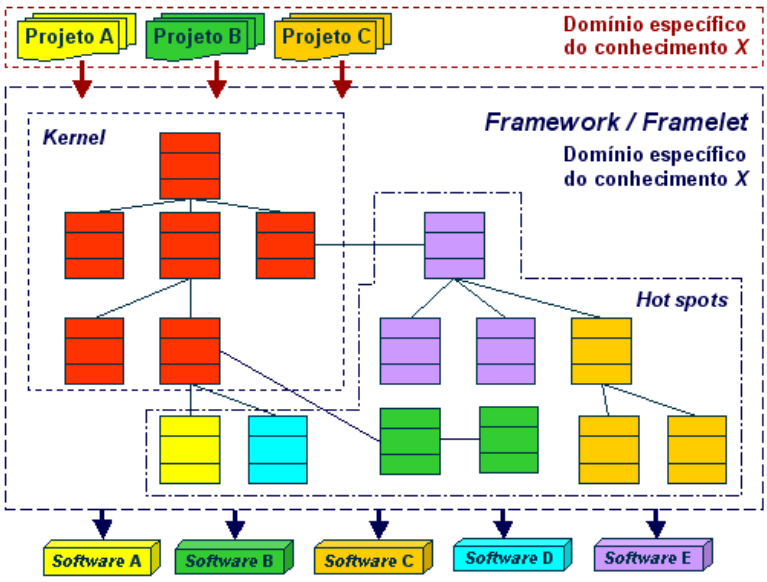


FIGURA 2.3 – Estrutura de um *Framework*.

A Fig. 2.3 ilustra a estrutura de um *Framework*. A partir da análise de diversas aplicações existentes (`Projeto A`, `Projeto B` e `Projeto C`) em um certo domínio de conhecimento (`Domínio X`), são extraídas as funcionalidades/recursos que todas possuem em comum. Elas constituirão o *kernel* do *Framework*. As funcionalidades/recursos que diferenciam essas aplicações constituirão os *hot spots*. A partir da estrutura resultante, é possível obter diversas aplicações semelhantes àquelas analisadas (`Software A`, `Software B` e `Software C`)

e, ainda, outras aplicações diferentes (`Software D` e `Software E`). O que irá diferenciar uma da outra é o conjunto de *hot spots* utilizados em cada instanciação do *Framework* (FAYAD, 1999a; CRESPO, 2000).

A estrutura apresentada na Fig. 2.3 pode ser compreendida através do exemplo ilustrado pela Fig. 2.4: um *Framework* para criação de editores gráficos.

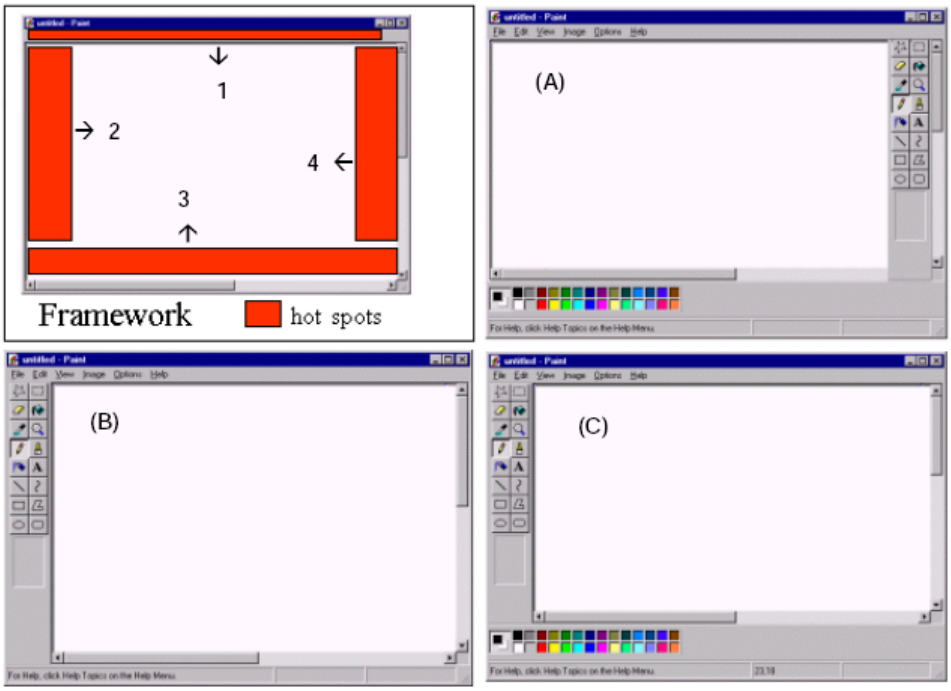


FIGURA 2.4 – Exemplo de *Framework* para criação de editores gráficos (RHEINHEIMER, 2002, p. 26).

As partes indicadas pelos números 1, 2, 3 e 4, na parte superior esquerda da figura, constituem os *hot spots* e, o restante, o *kernel*. Os *hot spots* indicam o que pode variar de um editor criado para outro, a saber: o editor pode apresentar um menu de comandos (indicado pelo número 1, na figura), uma barra de ferramentas para desenho (indicada pelos números 2 e 4, conforme seu posicionamento na tela) e uma barra de cores (indicada pelo número 3). O editor pode ter todos ou apenas alguns destes elementos.

As telas indicadas pelas letras A, B e C são exemplos de interfaces de editores gerados a partir do *Framework*: em `(A)`, temos um editor com menu, barra de cores e barra de

desenho alinhada à direita da tela, correspondente ao uso dos *hot spots* 1, 3 e 4; em `(B)`, temos um editor com menu e barra de desenho alinhada à esquerda da tela, relacionado ao uso dos *hot spots* 1 e 2; em `(C)`, temos um editor com menu, barra de desenho alinhada à esquerda e barra de cores, correspondente ao uso dos *hot spots* 1, 2 e 3 (RHEINHEIMER, 2002).

Pesquisas reconhecem a necessidade da integração de *Frameworks* e *Design Pattern*s, uma vez que os mesmos também promovem flexibilização e reutilização. Porém, não está claro como identificar os *hot spots* em *Framework*s, e como fazer a conexão entre a seleção do *Design Pattern* e a implementação destas variações (GAMMA, 1995; CRESPO, 2000; RHEINHEIMER, 2002).

2.2.1 Tipos de *Frameworks*

Há vários tipos de *Framework*s, nem todos OO, para os quais muitas formas de classificação têm sido propostas. Estes tipos podem variar de *Frameworks* de aplicação – que ajudam, por exemplo, a desenvolver interfaces de usuário – a *Frameworks* de mais baixo nível – que fornecem serviços básicos para comunicação, impressão e sistemas de arquivos. Existem também *Frameworks* de domínio específico, que solucionam problemas em áreas determinadas – como, por exemplo, acesso a dados, segurança e multimídia (FAYAD, 1999b, 1999c).

Frameworks de aplicação são classificados com base em seu escopo. Temos *Frameworks* de infraestrutura – que simplificam o desenvolvimento de uma infraestrutura de sistema portável e eficiente, como sistema operacional, comunicação, interfaces e processamento de linguagens –, *Frameworks* de integração *middleware* – comumente utilizados para integrar aplicações distribuídas e componentes – e *Frameworks* de negócios – relativos a domínios como telecomunicações, aviação, manufatura e finanças (FAYAD, 1999a, 1999b).

Um *Framework* também pode ser classificado quanto ao uso (FAYAD, 1999a, 1999b; CRESPO, 2000; MARKIEWICZ, 2000).

Assim, temos os tipos:

- *architecture-drive*n, ou *inheritance-focuse*d, ou *white box* – usado através da derivação de suas classes;
- *data-drive*n, ou *composition-focuse*d, ou *black box* – usado através da combinação de suas classes.

O tipo *whitebox* é mais difícil de usar, pois exige que o programador conheça a estrutura do *Framewor*k, mas também é considerado mais poderoso na mão de *expert*s. Possui classes incompletas, ou seja, que contém métodos sem codificação significativa. Para utilizar este tipo de *Framewor*k, programadores devem modificar seu comportamento utilizando herança para sobrescrever métodos de classes (FAYAD, 1999a, 1999b; MARKIEWICZ, 2000).

Já o tipo blackbox exige apenas que se saiba combinar as classes necessárias para realizar a instanciação. Possui componentes prontos para adaptação, e modificações são feitas através da composição de suas classes. É considerado o tipo ideal, pois é mais fácil utilizar um componente já existente do que construir um novo (FAYAD, 1999a, 1999b; MARKIEWICZ, 2000).

Outra classificação, apresentada por Fayad, Schmidt e Johnson (1999a), diz respeito ao modo como o *Framework* interage com outras aplicações:

- *called Framework* – é uma entidade passiva que pode ser chamada por outras aplicações. Corresponde a uma biblioteca de código e é utilizada através da chamada de funções ou métodos desta biblioteca;
- *calling Framework* – é uma entidade ativa que incorpora o controle dentro do próprio *Framewor*k. Aplicações fornecem métodos customizados ou componentes que são chamados pelo *Framewor*k.

Em geral, não se tem um *Framework* puramente *whitebox* ou *blackbo*x, ou puramente *called* ou *calling*, e sim uma combinação dos dois tipos. Algumas abstrações comuns entre vários componentes *blackbo*x, dentro de um mesmo domínio de problema, também podem ser

modeladas e implementadas utilizando herança e sobreposição de alguns métodos. Virtualmente, cada *Framework* é chamado por alguma parte da aplicação e também chama alguma outra parte (FAYAD, 1999a, 1999b).

2.2.2 Processos de desenvolvimento

Um *Framework* OO captura os aspectos comuns a uma família de aplicações. Ele também captura a experiência do projetista durante o processo de construção da aplicação. Isso possibilita reutilização tanto em *design* como em programação, o que não se consegue num processo de desenvolvimento OO tradicional (SCHMIDT, 1997).

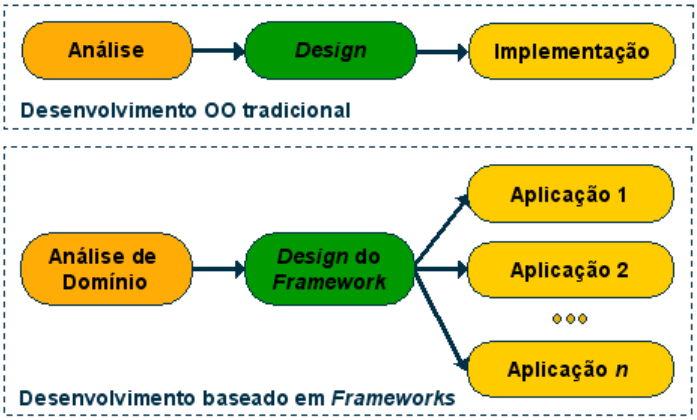


FIGURA 2.5 – Desenvolvimento OO tradicional *versus* baseado em *Frameworks*, adaptado de Crespo (2000, p. 28).

Como ilustra a Fig. 2.5, no processo tradicional a fase de análise fornece subsídios para a elaboração de um *design* que resultará na implementação de uma única aplicação. Já no processo baseado em *Frameworks*, a fase de análise do domínio define requisitos que resultarão no *design* de um *Framework* que pode ser instanciado diversas vezes, gerando diferentes aplicações pertencentes ao domínio analisado. Isto faz com que não seja necessário um novo trabalho de análise e *design* para construir uma nova aplicação: apenas deve-se customizar os

hot spots que constituirão as características desejadas para o *softwar*e. Desta forma, *Frameworks* proporcionam reuso em três níveis: análise, projeto e implementação (FAYAD, 1999a; MATTSON, 1999, 2000 *apud* CRESPO, 2000, p. 30; MARKIEWICZ, 2000).

O processo de desenvolvimento de um *Framework* é mais difícil e trabalhoso, pois são estudados problemas de um determinado domínio e observadas características de diversas aplicações pertencentes a este domínio. O conjunto de requisitos definidos deve ser o mais completo possível, para garantir que o *Framework* funcione corretamente (CRESPO, 1998; FAYAD, 1999a).

Além disso, toma mais tempo devido às diversas iterações de modelagem, codificação, testes e correções, necessárias para garantir o funcionamento adequado do *Framework* (FAYAD, 1999a).

Em geral, a primeira versão de um *Framework* é *whitebo*x. A experiência leva a melhoramentos no *Framework* que, quase sempre, vai se tornando mais e mais *blackbox* – para isto, é fundamental que ele seja utilizado, testado. Finalmente, chega o momento em que o *Framework* é considerado pronto (FAYAD, 1999a).

O desenvolvimento de um *Framework* pode ser realizado de algumas formas, dependendo daquilo em que se baseia: experiência, análise de domínio ou *Design Pattern*s.

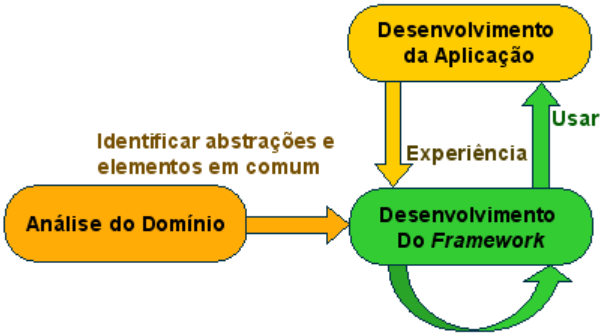


FIGURA 2.6 – Desenvolvimento de *Frameworks* baseado em experiência, adaptado de Crespo (2000, p. 30).

A Fig. 2.6 ilustra o desenvolvimento baseado em experiência, que é iniciado desenvolvendo-se *n* aplicações – no mínimo, duas – baseadas no domínio do problema. Quando prontas, observa-se suas características em comum, colocando-as no *Framewor*k.

Para verificar se as características extraídas estão corretas, refaz-se as aplicações com base no *Framework* desenvolvido. No caso de demandar muito esforço, reescreve-se o *Framework* quando necessário e utiliza-se essa experiência no desenvolvimento de sua nova versão. Desenvolvem-se outras aplicações com o *Framework* e repetem-se as interações quantas vezes forem necessárias (MATTSON, 1999, 2000 *apud* CRESPO, 2000, p. 29).

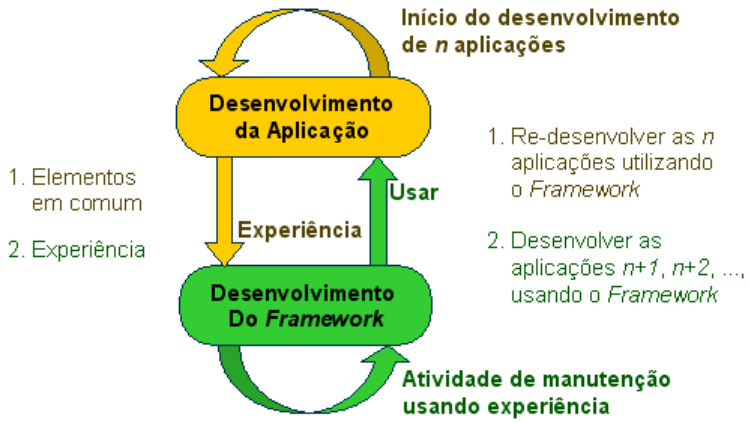


FIGURA 2.7 – Desenvolvimento de *Frameworks* baseado na análise de domínio, adaptado de Crespo (2000, p. 30).

O processo baseado em análise do domínio – mostrado na Fig. 2.7 – tem, como primeira etapa, a análise do domínio do problema para identificar e entender possíveis abstrações Analisar o domínio requer analisar aplicações existentes e, segundo Fayad, Schmidt e Johnson (1999a), consiste de três atividades maiores:

- análise do contexto do domínio – identificar o escopo do domínio;
- modelagem das abstrações do domínio – identificar os componentes certos do domínio e seus dados, comportamento e interação;
- arquitetura do domínio – gerar o *backbone* do *Framewor*k.

Após identificar as abstrações, desenvolve-se o *Framework* juntamente com uma aplicação de teste, modificando o *Framework* quando necessário, revisando as aplicações anteriores para ver se continuam funcionando (CRESPO, 1998; FAYAD, 1999a).

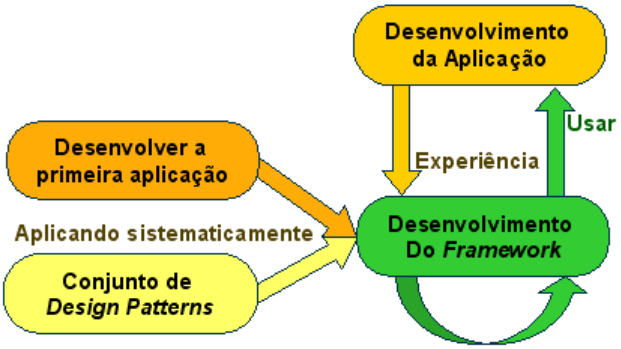


FIGURA 2.8 – Desenvolvimento de *Frameworks* utilizando *Design Patterns*, adaptado de Mattson (1996, p. 62).

A Fig. 2.8 ilustra o desenvolvimento baseado em *Design Patterns*. O primeiro passo consiste em desenvolver uma aplicação e em seguida aplicar sistematicamente um conjunto de padrões para criar o *Framework*. A partir daí, ocorrem interações entre aplicações e *Framework* (CRESPO, 1998; MATTSON, 1999, 2000 *apud* CRESPO, 2000, p. 31).

Design Patterns mapeiam características estruturais de um *Framework* e podem, em adição, proporcionar inspiração na busca dos *hot spots*. Também documentam aspectos mais técnicos do *design*: a solução que oferecem está ligada a um contexto e problema, proporcionando documentação para "o que" e "por que" do *design*. Proporcionam flexibilidade, facilitando o reuso e as modificações, contribuindo para a diminuição das iterações necessárias até que o *Framework* seja considerado pronto (FAYAD, 1999a; MARKIEWICZ, 2000).

Na Figura 2.9, temos o que seria um processo geral de desenvolvimento.

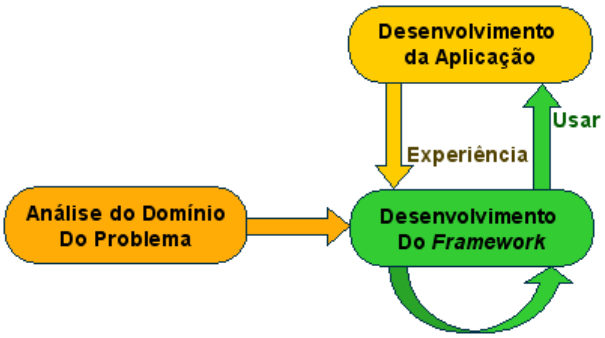


FIGURA 2.9 – Processo geral de desenvolvimento de *Frameworks*, adaptado de Crespo (2000, p. 29).

Este processo, segundo apresentado por Crespo (1998) reúne elementos em comum dos outros processos de desenvolvimento descritos:

- é feita a análise de domínio do problema;
- a primeira versão do *Framework* é desenvolvida utilizando as abstrações encontradas;
- uma ou mais aplicações são desenvolvidas baseadas no *Framewor*k. O teste é importante para verificar se o *Framework* é realmente reutilizável;
- problemas encontrados no desenvolvimento da aplicação baseado no *Framework* são capturados e resolvidos na próxima versão.

Após repetir o ciclo várias vezes, o *Framework* vai atingindo um nível de maturidade aceitável, possibilitando sua reutilização por vários usuários.

2.3 ONTOLOGIAS

O termo “ontologia” vem da Filosofia. É uma disciplina que estuda a natureza da existência: que tipos de coisas existem e como elas se relacionam (BERNERS-LEE, 2001; ZÚÑIGA, 2001; GRUBER, 1993).

Gruber foi quem originou o termo na área da Inteligência Artificial, motivado por razões como compartilhamento e reuso do conhecimento. A idéia de Ontologia, nesse contexto, é de “especificação explícita de uma conceitualização”. Ou seja: um vocabulário comum, utilizado de forma coerente e consistente para definir objetos, conceitos e outras entidades existentes em um domínio de conhecimento, bem como o relacionamento entre eles (GRUBER, 1993; BENCH-CAPON, 1997; CUI, 1999).

Esta não é a única definição encontrada na literatura. Guarino reescreve a definição de Gruber: “uma ontologia é uma teoria lógica que descreve o significado pretendido para um vocabulário formal, i. e. seu compromisso ontológico com uma conceitualização específica do

mundo". Há ainda definições de outros autores, mas Gruber e Guarino são os mais citados (CUI, 1999).

2.3.1 Representação de Ontologias

A maioria das linguagens propostas utiliza a lógica para representação de Ontologias. Descrita desta forma, uma Ontologia consiste de um conjunto de sentenças – cada uma com um significado que não deve ser ambíguo – combinadas através de operadores lógicos. Porém, computacionalmente, a representação lógica não é fácil de ler e processar. Assim, na maioria das linguagens, foram desenvolvidos formalismos adicionais para representar estruturas. Estes formalismos atingem diversos níveis, desde altamente informais – por exemplo, linguagem natural – até rigorosamente formais (CUI, 1999).

Alguns recursos específicos para a descrição de Ontologias são:

- OilEd[13] – editor de Ontologias que utiliza a linguagem *DARPA Agent Markup Language + Ontology Inference Layer*[14] (*DAML+OIL*). Permite a verificação de consistência das Ontologias;
- Ontolingua[15] – ferramenta para compartilhamento e reuso de informação. É baseada em um formato chamado *Knowledge Interchange Format* (*KIF*) e visa facilitar reuso e interoperabilidade. Possui diversas Ontologias que podem ser incorporadas àquelas que vão sendo escritas;
- *Web Ontology Language*[16] (*OWL*) – linguagem de descrição de Ontologias para a *Web*, baseada em *XML*;

[13] Informações podem ser encontradas em: <http://oiled.man.ac.uk/>
[14] Informações podem ser encontradas em: <http://www.w3.org/TR/daml+oil-reference/>
[15] Informações podem ser encontradas em: <http://www.ksl.stanford.edu/software/ontolingua/>
[16] Informações podem ser encontradas em: <http://www.w3.org/TR/owl-reference/>

- OntoEdit[17] – permite a edição e manutenção de Ontologias de forma gráfica. Suporta *F-Logic*, apresentada por Kifer (1990), *Resource Description Framework Schema*[18] (*RDF-Schema*) e *OIL*[19]. Possibilita a exportação das Ontologias criadas para outros formatos como, por exemplo, *Document Type Definition*[20] (*DTD*).
- *Protègè* 2000[21] – ferramenta integrada, com interface gráfica, que permite o desenvolvimento de sistemas baseados em conhecimento. Uma de suas funcionalidades é a modelagem de Ontologias. Possui alguns exemplos que podem ser consultados pelo usuário, como a Ontologia de um jornal (*Newspaper*), mostrada na Fig. 2.10.

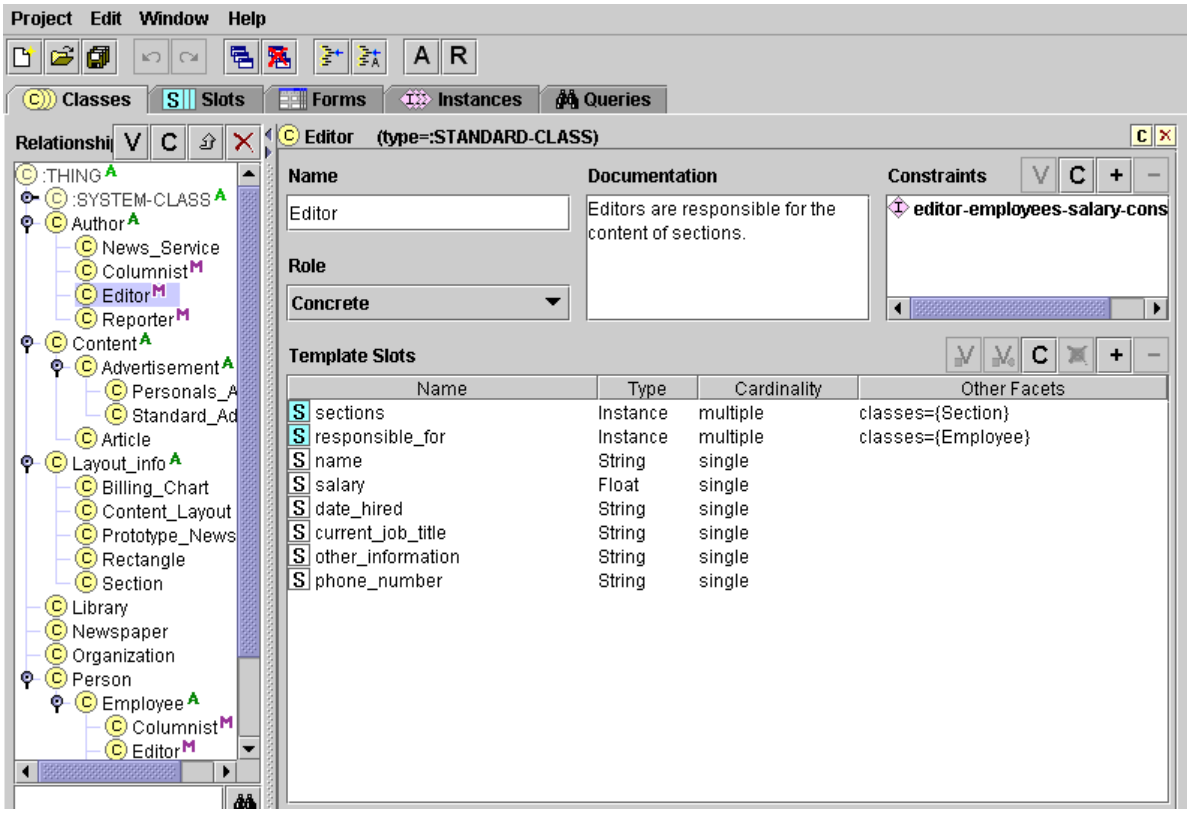


FIGURA 2.10 – Exemplo de Ontologia (*Newspaper*) criada no *Protègè* 2000.

[17] Informações podem ser encontradas em: <http://www.ontoknowledge.org/tools/ontoedit.shtml>
[18] Informações podem ser encontradas em: <http://www.w3.org/TR/rdf-schema/>
[19] Informações podem ser encontradas em: <http://www.ontoknowledge.org/oil/>
[20] Informações podem ser encontradas em: <http://www.w3.org/TR/REC-xml/>
[21] Informações podem ser encontradas em: <http://protege.stanford.edu/>

A Fig. 2.10 também mostra a forma mais típica de representação de uma Ontologia: a forma taxonômica. A taxonomia define classes de objetos e sua hierarquia, ou seja, é baseada em generalização e especialização. Relacionamentos entre as entidades podem ser representados através da definição de propriedades das classes. Caso as classes possuam subclasses, as propriedades são herdadas (BERNERS-LEE, 2001).

Assim, a Ontologia da Fig. 2.10 mostra diversas classes, que representam o que constitui um jornal: autores (indicado, na parte esquerda da tela, pela classe `Author`), conteúdo (classe `Content`), elementos de *layout* (classe `Layout_info`), biblioteca (classe `Library`), jornais publicados (classe `Newspaper`), pessoas (classe `Person`), e outros. Cada uma dessas classes pode conter outras, indicando classificação: por exemplo, um autor pode ser um editor (classe `Editor`, que é subclasse de `Author`), repórter (subclasse `Reporter`), colunista (subclasse `Columnist`), etc. As classes apresentadas apresentam algumas propriedades, indicadas na parte direita da tela: por exemplo, um editor possui nome (`name`), salário (`salary`), data de admissão (`date_hired`), etc. Vínculos entre classes também podem ser definidos e, no ambiente Protègè, são registrados como propriedades: por exemplo, um editor está vinculado a sessões do jornal (propriedade `sections`, que faz referência à classe `Section`) e é responsável por funcionários (propriedade `responsibe_for`, que faz referência à classe `Employee`).

A partir da iniciativa de Gruber, esforços têm sido feitos para facilitar a criação e manutenção de Ontologias em algum formato de representação, bem como sua utilização no desenvolvimento de sistemas de diversas áreas. Esforços também surgiram no sentido de definir técnicas de elaboração de Ontologias, como pode ser visto a seguir.

2.3.2 Metodologias de desenvolvimento

Uschold e Grüninger (1996) propõem técnicas manuais para a construção de Ontologias.

Os autores descrevem um roteiro geral:

- identificação de propósito e escopo – deixar claro o motivo pelo qual a Ontologia está sendo desenvolvida e o uso ao qual se destina;
- construção da Ontologia – este processo consiste das fases de captura (definição de conceitos chave e relacionamentos no domínio de interesse, bem como a produção de representações textuais precisas e não ambíguas para esses conceitos e relacionamentos, com a identificação de termos que se referem aos mesmos), codificação (representação explícita da conceitualização obtida na fase anterior, utilizando alguma linguagem formal) e integração de Ontologias existentes (durante a fase de captura ou codificação, identificar Ontologias relacionadas que já existam, utilizando-as para compor parte da Ontologia em desenvolvimento ou para servir de base para a criação da nova Ontologia);
- avaliação – julgamento técnico da Ontologia desenvolvida (levando em consideração, entre outros aspectos, o ambiente de *software* a que será associada);
- documentação – os conceitos principais e as primitivas utilizadas para expressar definições devem estar bem documentados. Como é reconhecido pelos autores, o uso de ferramentas como Ontolingua facilita o desenvolvimento de Ontologias, em especial a sua documentação.

Diversas técnicas foram desenvolvidas para o desenvolvimento de Ontologias, baseadas ou não na utilização de ferramentas. No trabalho de Jones, Bench-Capon e Visser (1998), encontramos a descrição de muitas dessas metodologias, entre elas: Tove e Methontology.

Um exemplo mais recente pode ser encontrado em um artigo de Noy e McGuiness (2002), onde são definidos os passos básicos para a criação de uma Ontologia. O roteiro é semelhante àquele descrito por Uschold e Grüninger (1996): definir o domínio e o escopo, verificar se é possível reutilizar Ontologias já existentes, listar termos importantes, definir

classes, identificar a hierarquia das classes e definir as propriedades das classes. A diferença é que o roteiro é proposto no contexto da utilização do ambiente *Protègè* 2000.

Ontologias também estão sendo utilizadas na *Web* e fazem parte do contexto da *Web* Semântica.

2.3.3 Ontologias e *Web* Semântica

Em meados de 2000, a visão de uma *Internet* semântica passou a ser apregoada por um dos idealizadores da *Internet*: Tim Bernes-Lee (FREITAS, 2002).

A *Web* Semântica, segundo Berners-Lee (2001), é uma extensão da *Web* onde as informações são disponibilizadas de uma forma bem definida, facilitando a cooperação entre humanos e computadores. O objetivo é fazer com que computadores sejam capazes de melhor representar e entender dados, e não somente exibi-los. Isso é feito através de uma linguagem que expressa tanto dados quanto regras para inferência, ou seja, adiciona lógica à *Web*.

Há três perspectivas para a *Web* Semântica, que levam a diferentes expectativas quanto à sua contribuição para a *Web* que temos hoje (MARSHALL, 2003). São elas:

- intenção, do *W3C*, de trazer ordem às redes de documentos fracamente relacionadas que compõem a *Web*;
- idealização, por Tim Berners-Lee, da criação de uma base de conhecimento globalmente distribuída;
- visão de criar uma estrutura para o compartilhamento coordenado de dados e conhecimento.

Apesar de a *Web* Semântica ser uma visão futura da *Web*, algumas tecnologias importantes para a sua criação já existem. O *W3C* vem trabalhando na *OWL*, que utiliza as facilidades da *XML* para definição de *tags* e *schemas*, bem como a flexibilidade para

representação de dados proporcionada pelo *RDF*[22] (BERNERS-LEE, 2001; W3C WEB SERVICES ARCHITECTURE WORKING GROUP, 2003).

2.3.4 Áreas de aplicação

Segundo Uschold e Grüninger (1996), Ontologias contribuem para a redução de confusão conceitual e terminológica, proporcionando um entendimento compartilhado. Este entendimento serve de base para:

- comunicação entre pessoas com diferentes necessidades e pontos de vista, devido aos diferentes contextos em que se encontram;
- interoperabilidade entre sistemas, obtida através da tradução de informações entre diferentes métodos de modelagem, paradigmas, linguagens e ferramentas de *software*;
- benefícios para a engenharia de sistemas, em particular: reusabilidade da representação do conhecimento compartilhado; confiabilidade gerada pela possibilidade de automatizar a checagem de consistência de uma representação formal; especificação (o conhecimento compartilhado pode ajudar na identificação de requisitos e na definição da especificação de um sistema).

Aplicações de Ontologias, de forma geral, podem ser classificadas em quatro categorias principais, sendo que uma aplicação pode pertencer a mais de uma categoria (JASPER, 1999; FALBO, 2002). Essas categorias são:

- autoria neutra – uma Ontologia é desenvolvida em uma única linguagem e pode, então, ser traduzida para diferentes formatos e utilizada em diferentes aplicações;
- ontologia como especificação – uma Ontologia, referente a um certo domínio, é criada e fornece um vocabulário a ser utilizado para definição de requisitos para

[22] Informações podem ser encontradas em: <http://www.w3.org/TR/rdf-primer/>

uma ou mais aplicações. Neste sentido, uma Ontologia pode ser vista como um modelo de domínio e permite reuso de conhecimento, pois é utilizada como base para especificação e desenvolvimento de diversas aplicações;

- acesso comum a informações – uma Ontologia é aplicada para permitir que diversos Agentes (ou humanos) tenham acesso a fontes heterogêneas de informação, que são expressadas em vocabulários diversos ou em formato inacessível;
- pesquisa baseada em Ontologias – uma Ontologia é utilizada para realizar buscas em um repositório de informações, aumentando a precisão e reduzindo o tempo global de busca.

O uso de Ontologias vem sendo pesquisado e aplicado em diversos campos. Alguns exemplos são: sistemas da área jurídica (BENCH-CAPON, 1997), bancos de dados (CASTANO, 1999; HAKIMPOUR, 2001), sistemas médicos (BURGUN, 2001), segurança de informações (RASKIN, 2002), ensino (CAMPOS, 2002; GOÑI, 2002), engenharia de *software* (DEVEDZICK, 1999; FALBO, 2002; FELICÍSSIMO, 2003; SHANKS, 2003), Agentes (MAAMAR, 1999, ESTOMBELO-MONTESCO, 2003; GROSOF, 2003; PAN, 2003), negócios (CUI, 1999; GLUSHKO, 1999; GRÜNINGER, 2000; WILLIAMS, 2003) e *Web Services* (BUSSLER, 2002; GIBBINS, 2003; PAHL, 2003; TSAI, 2003).

2.4 AGENTES DE *SOFTWARE*

O advento dos Agentes de *Software* trouxe à tona diversas discussões sobre o que um Agente é e como difere dos programas em geral (FRANKLIN, 1996).

A noção de um Agente tem raízes na expressão "*agens*" do Latin, que significa: a causa de um efeito, uma substância ativa, uma pessoa ou uma coisa que age, ou um representante (TOKORO, 1994).

Alan Kay – um dos proponentes da tecnologia de Agentes – diz:

> A idéia de um agente teve origem com John McCarthy na metade da década de 50, e o termo foi criado por Oliver G. Selfridge poucos anos antes, quando ambos estavam no *Massachusetts Institute of Technology*. Eles tinham em mente um sistema que, quando definido um objetivo, pudesse resolver os detalhes das operações computacionais apropriadas e pudesse pedir e receber conselhos, oferecidos em termos humanos, quando estivesse em dificuldade. Um agente poderia ser um '*soft robot*' vivendo e realizando suas tarefas dentro do mundo dos computadores (KAY, 1984 *apud* BRADSHAW, 1997, p. 4).

Nwana (1996 *apud* BRADSHAW, 1997, p. 4) divide a pesquisa de Agentes em duas correntes principais:

- a primeira, que teve início em 1977, tem raízes na inteligência artificial distribuída. Esta corrente concentra-se principalmente em Agentes deliberativos com modelos internos simbólicos, o que contribuiu para o entendimento de aspectos como interação e comunicação entre Agentes, decomposição e distribuição de tarefas, coordenação e cooperação, resolução de conflitos através de negociação, etc;
- a segunda, com início em 1990, é um movimento mais recente e que veio crescendo rapidamente. Esta corrente estuda uma gama muito mais ampla de Agentes, com diversos níveis de inteligência. Em contraste com a primeira corrente, a ênfase muda da reflexão para a ação, do raciocínio para a ação remota.

O uso de Agentes foi adotado por diversas áreas ao longo do tempo – inteligência artificial, robótica, computação distribuída, redes de computadores, computação gráfica, entre outros –, gerando uma grande diversidade de aplicações e abordagens. Isso levou ao uso do termo "agente" sem um consenso sobre seu significado, situação que se estende até os dias de hoje.

Um exemplo de Agente, utilizado no dia a dia de muitas pessoas, é o assistente do editor de textos *Microsoft Word*, mostrado na Fig. 2.11. O assistente visa ajudar o usuário no uso do editor, oferecendo orientações sobre como realizar tarefas (relacionadas à impressão de

documentos, por exemplo) e pode ser configurado para assumir a aparência desejada pelo usuário (um clip, por exemplo). Ele também pode ser ocultado, caso o usuário não deseje ajuda.

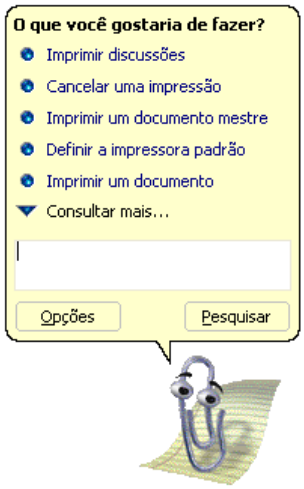


FIGURA 2.11 – Exemplo de Agente: assistente do *Microsoft Word*.

Diversas definições de Agente(s) podem ser encontradas na literatura, entre elas:

- "(...) programas que participam de diálogos [e] negociam e coordenam transferência de informação" (COEN *apud* FRANKLIN, 1996);
- "(...) alguma coisa que pode ser vista como percebendo seu ambiente através de sensores e agindo sobre esse ambiente através de executores" (RUSSEL, 1995);
- "(...) um sistema situado em e parte de um ambiente, que percebe esse ambiente e atua sobre ele no tempo, em cumprimento de sua própria agenda e causando efeito no que ele venha a perceber no futuro" (FRANKLIN, 1996);
- "(...) componentes (de *software*) ativos, persistentes, que percebem, raciocinam, agem e comunicam" (HUHNS, 1998 *apud* WAGNER, D. N., 2000).

Smith define um Agente como:

> (...) uma entidade de *software* persistente dedicada a um propósito específico. "Persistente" distingue Agentes de subrotinas; Agentes têm suas próprias idéias sobre como executar tarefas, suas próprias agendas. "Propósito específico" os distingue de aplicações multifuncionais; Agentes são tipicamente muito menores (SMITH, 1994 *apud* FRANKLIN, 1996).

Segundo definição da *IBM*, Agentes são:

> (...) entidades de *software* que realizam um conjunto de operações em favor de um usuário ou outro programa com um certo grau de independência ou autonomia e, fazendo isso, empregam algum conhecimento ou representação das metas ou desejos do usuário (INTERNATIONAL BUSINESS MACHINES *apud* FRANKLIN, 1996).

Franklin e Graesser (1996) apresentam sua definição como a essência do que é ser um Agente: estar inserido em um ambiente e fazer parte dele; perceber o ambiente e agir sobre ele de forma autônoma; não necessitar de outras entidades para fornecer entradas ou interpretar e utilizar saídas; agir em cumprimento de sua própria agenda, ou cumprir metas definidas por algum outro Agente; agir de forma tal que suas ações presentes possam afetar sua percepção futura, afetar seu ambiente; agir continuamente durante algum período de tempo.

Segundo os autores, Agentes de *Software* são programas por definição, mas um programa precisa satisfazer muitos requisitos até poder ser considerado um Agente.

Na área de Engenharia de *Software*, o uso de Agentes no desenvolvimento de sistemas é algo relativamente recente e tem sido tema de crescente estudo. O que se busca é aplicar teorias de Agentes para superar as limitações da Engenharia de *Software* Orientada a Objetos e melhor lidar com a questão da complexidade no processo de desenvolvimento de *software* (GARCIA, 2003; CUESTA-MORALES, 2004; ZAMBONELLI, 2004).

Assim, está surgindo a Engenharia de *Software* Orientada a Agentes. Dentro desta nova abordagem, um tema de grande discussão é “Agentes *versus* Componentes de *Software*” (AMOR, 2000; LIND, 2001; KRUTISCH, 2003).

A seguir, será apresentada uma série de conceitos que oferecem uma visão geral sobre Agentes e, mais especificamente, Agentes de *Software*. Ao final, será abordada a questão “Agentes *versus* Componentes de *Software*”.

2.4.1 O que caracteriza um Agente?

Algumas propriedades fundamentais, que diferenciam Agentes de programas em geral, são definidas por Wooldridge e Jennings (1995):

- autonomia: Agentes operam sem a intervenção direta de humanos ou outros, e têm algum tipo de controle sobre suas ações e estado interno;
- habilidade social: Agentes interagem com outros Agentes (e possivelmente humanos) através de algum tipo de linguagem para comunicação entre Agentes. Alguns exemplos deste tipo de linguagem são a *Agent Communication Language*[23] (*ACL*) e a *Knowledge Query and Manipulation Language*[24] (*KQML*);
- reatividade: Agentes percebem seu ambiente (que pode ser o mundo físico, um usuário via uma interface gráfica, uma coleção de outros Agentes, a *Internet*, ou talvez todos esses ambientes combinados) e respondem de maneira adequada às mudanças que nele ocorrem;
- pró-atividade: Agentes não agem simplesmente em resposta ao seu ambiente, eles são capazes de exibir um comportamento direcionado a objetivos, de tomar a iniciativa.

A Tab. 2.1 mostra mais algumas propriedades, sobre as quais falam Franklin e Graesser (1996).

TABELA 2.1 – Algumas propriedades de Agentes, adaptado de Franklin (1996).

Propriedade	Significado
Continuidade temporal	Um processo continuamente em execução
Aprendizado, adaptatividade	Mudança de comportamento com base em experiências anteriores
Mobilidade	Habilidade de transportar-se de uma máquina para outra
Flexibilidade	As ações não possuem um roteiro fixo
Personalidade	Presença de personalidade e estado emocional

[23] Informações podem ser encontradas em: <http://www.fipa.org/repository/aclspecs.html>
[24] Informações podem ser encontradas em: <http://www.cs.umbc.edu/kqml/>

Ainda, Agentes podem: fazer parte do mundo real (um robô) ou de um ambiente computacional (um Agente de *Software*); apresentar níveis de inteligência (capacidade de realizar metas com base em conhecimento e raciocínio); ter uma representação gráfica (FRANKLIN, 1996; BRADSHAW, 1997; SANTOS, 2003).

2.4.2 Tipos de Agentes

Agentes podem ser classificados segundo suas propriedades – levando em conta uma única propriedade ou, então, a combinação de algumas delas. Cabe ressaltar que, mesmo que um Agente seja dito móvel, ele também satisfaz as propriedades de autonomia, habilidade social, reatividade, pró-atividade e continuidade temporal (FRANKLIN, 1996).

A fim de caracterizar tipos de Agentes de maneira mais simples do que tentar descrever todas as possíveis combinações de propriedades, pesquisadores criaram diversas classificações e taxonomias (BRADSHAW, 1997).

Russel e Norvig (1995) classificam Agentes em quatro tipos, segundo aspectos diferenciados na busca da solução de problemas:

- reflexivo – cada condição ou percepção dispara alguma ação pré-estabelecida (regra de condição-ação);
- reflexivo que mantém registro do ambiente – versão melhorada do tipo anterior, onde o estado interno do Agente é atualizado (ocorre um registro do estado atual do ambiente);
- baseado em metas – o Agente tem idéia do estado atual do ambiente, bem como de uma meta (estado desejável a ser atingido). Ele então faz uma combinação do desejável com o resultado de possíveis ações, para tomar decisões na busca da meta. A meta pode ser alcançada com uma única ação ou até com longas seqüências de ações;

- baseado na utilidade – semelhante ao tipo anterior, porém é indicada a preferência por um certo estado, que passa a ser o mais "útil" para o Agente. A utilidade facilita a tomada de decisões em duas situações: primeiro, quando há metas conflitantes e somente uma pode ser atingida; segundo, quando há diversas metas e nenhuma pode ser atingida de forma adequada (a utilidade ajuda na busca do melhor possível para este caso).

Na área da inteligência artificial distribuída, Moulin e Chaib-Draa (1996 *apud* BRADSHAW, 1997, p. 8) classificam Agentes segundo o seu grau de capacidade para resolver problemas.

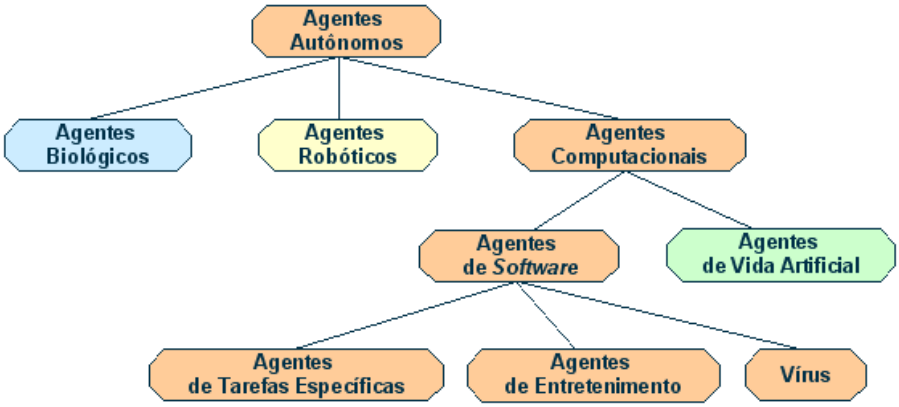


FIGURA 2.12 – Classificação de Agentes, adaptado de Franklin (1996).

Franklin e Graesser (1996 *apud* BRADSHAW, 1997, p. 11) oferecem a taxonomia mostrada na Fig. 2.12 como aquela que cobre muitos dos exemplos encontrados na literatura. Abaixo desta classificação inicial, eles sugerem que Agentes sejam categorizados por estruturas de controle, ambientes (banco de dados, sistema de arquivos, rede, *Internet*), linguagem em que são escritos e aplicações.

Nwana (1996 *apud* BRADSHAW, 1997, p. 9) propõe uma tipologia de Agentes que identifica outras dimensões de classificação. Agentes podem ser classificados segundo:

- mobilidade – estáticos ou móveis;
- presença de um modelo de raciocínio simbólico – deliberativos ou reativos;
- papéis – de informação ou *Internet*;

- exibição de atributos ideais e principais, como cooperação, autonomia e aprendizado – colaborativos, de aprendizado colaborativo, interfaces e espertos;
- filosofias híbridas, que combinam duas ou mais abordagens em um único Agente;
- atributos secundários, como versatilidade, benevolência, veracidade, continuidade temporal, qualidades mentais e emocionais, etc.

Wooldridge e Jennings (1996 *apud* SANTOS, 2003, p. 41) propõem uma classificação segundo o nível de sofisticação dos Agentes:

- *gopher* – executam tarefas baseadas em regras pré-especificadas;
- *service performing* – executam tarefas a partir de requisições de usuários;
- *predictive* – disponibilizam informações ou executam ações para o usuário de forma voluntária, até certo ponto.

Rickel e Johnson (1997 *apud* SANTOS, 2003, p. 41) falam sobre Agentes pedagógicos, que atuam em ambientes educacionais.

Maglio e Barret (2000) discutem Agentes intermediários, que transformam significativamente a informação à medida que ela trafega de uma máquina para outra. Agentes intermediários, se colocados em meio ao fluxo de informações trocadas entre produtor e consumidor, podem personalizar essas informações de acordo com pessoas, dispositivos e situações.

Bezek e Gams (2003) definem três tipos de Agentes, segundo a interação que realizam:

- reativo a Agentes – somente respondem à demanda de outros Agentes;
- reativo ao sistema – utiliza informações obtidas de outros Agentes, mas não fornece informação. Sua tarefa principal é monitorar o sistema de Agentes e reagir de acordo com seus objetivos;
- colaborativo – sincroniza suas ações de acordo com requisições e respostas do sistema de Agentes. Suas ações podem ser iniciadas ou canceladas de acordo com as ações de outros Agentes.

Existem outras classificações que podem ser encontradas na literatura, muitas delas combinando tipos aqui citados.

Alguns pesquisadores da área de inteligência artificial também classificam Agentes como fortes ou fracos na sua essência, de acordo com as propriedades que apresentam, principalmente quanto a conhecimento e raciocínio, e o tipo de tarefa que realizam (BRADSHAW, 1997).

2.4.3 Arquiteturas

As arquiteturas de Agentes podem ser classificadas de acordo com o tipo de Agentes que a compõem e a finalidade para que se destinam, sugerindo uma metodologia de implementação a ser seguida.

Knapik e Johnson (1998 *apud* FERREIRA, 2002, p. 6) apresentam a seguinte classificação, onde uma arquitetura pode ser:

- simples – um único Agente;
- moderada – Agentes que realizam as mesmas tarefas, mas possuem diferentes usuários e podem residir em máquinas diferentes;
- complexa – diferentes tipos de Agentes, cada um com certa autonomia, que podem cooperar e estar em diferentes plataformas.

Segundo Wooldridge e Jennings (1995), uma arquitetura pode ser:

- deliberativa – Agentes possuem um modelo simbólico do ambiente, explicitamente representado. Suas ações são decididas por raciocínio lógico. Constitui a abordagem clássica da Inteligência Artificial;
- reativa – Agentes não possuem um modelo do ambiente e não utilizam raciocínio lógico; devem desenvolver inteligência a partir de suas interações com o ambiente, sem a necessidade de um modelo pré-estabelecido;

- híbrida – é um misto das duas arquiteturas citadas anteriormente. Possui um subsistema deliberativo (que planeja e toma decisões de maneira simbólica) e um reativo (que reage a eventos ocorridos no ambiente sem ocupar-se de raciocínios complexos). Busca, desta forma, ser mais adequada e funcional para a construção dos Agentes.

Outra classificação é proposta por Weiss (1999 *apud* FERREIRA, 2002, p. 10):

- reativa – Agentes tomam decisões com base em alguma forma de mapeamento direto da situação para a ação;
- baseada em lógica – Agentes tomam decisões através de dedução lógica;
- híbrida – Agentes tomam decisões por meio de várias camadas de *software*, onde cada uma pode raciocinar sobre o ambiente em diferentes níveis de abstração;
- de crença-desejo-intenção – Agentes tomam decisões através da manipulação de estruturas de dados que representam suas crenças, desejos e intenções.

Há ainda outras classificações encontradas na literatura. Entre elas, há aquelas que levam em conta aspectos como coordenação e cooperação entre os Agentes (FERREIRA, 2002).

A cooperação vem da necessidade de Agentes obterem ajuda de outros para realizar tarefas. As principais arquiteturas neste contexto, apresentadas por Ferreira (2002), são:

- quadro negro – Agentes não se comunicam diretamente, mas sim através de um "quadro negro" que serve de repositório para perguntas e respostas. Esta arquitetura não se mostra adequada para sistemas de tempo real e em que o tempo de resposta adequado seja fundamental para garantir o funcionamento desejado;
- troca de mensagens – Agentes comunicam-se diretamente por meio de mensagens assíncronas, com uso de *broadcasting*. Em lugar do quadro negro, pode haver um Agente facilitador da comunicação. É preciso definir um protocolo de comunicação (formalismo necessário para que os Agentes troquem mensagens e as compreendam). Ainda, cada Agente deve conhecer o endereço dos demais para poder comunicar-se.

Nesta arquitetura, as mensagens são obtidas em tempo hábil, mas sua implementação é dificultada quando há um crescimento exponencial de mensagens trocadas no ambiente;

- federativa – Agentes são agrupados segundo um critério escolhido. Junto aos grupos, há Agentes facilitadores que são encarregados de receber mensagens e encaminhá-las a um Agente destinatário, caso ele esteja presente no grupo. Esta arquitetura busca diminuir o fluxo de mensagens desnecessárias no ambiente.

A coordenação traduz a forma como os Agentes estão organizados para cooperar para alcançar um objetivo comum no sistema. Conforme apresentado por Ferreira (2002), uma arquitetura, neste contexto, pode fazer uso dos seguintes mecanismos:

- mestre-escravo – Agentes podem ser gerentes (mestres) ou trabalhadores (escravos). Os trabalhadores são coordenados por um gerente que distribui as tarefas e aguarda o resultado. Caso haja Agentes em grupos, pode haver a figura do facilitador;
- de mercado – Agentes ocupam um mesmo nível e sabem as tarefas que cada um é capaz de desempenhar. Este mecanismo visa diminuir a quantidade de mensagens trocadas no ambiente, visto que cada Agente já conhece os outros.

2.4.4 Áreas de aplicação

Agentes de *Software* vêm adquirindo uma crescente relevância em pesquisa e aplicações (WAGNER, D. N., 2000). Em trabalhos de Jennings e Wooldridge (1996, 1998), são mostradas algumas áreas principais em que Agentes podem ser aplicados:

- indústria – manufatura, controle de processos, telecomunicações, gerenciamento de informações, gerenciamento de redes, controle de tráfego aéreo, sistemas de transporte;
- comércio – gerenciamento de informações e processos de negócio, comércio eletrônico;

- entretenimento – jogos, cinema e teatro interativos;
- saúde – monitoração de pacientes, sistemas de atenção à saúde.

Wagner (D. N., 2000) também cita as seguintes aplicações: *Data Mining*, educação, bibliotecas digitais, *Personal Digital Assistants* (*PDAs*, também conhecidos como *palmtops*).

Uma área de aplicação recente é a da *Web* Semântica (BERNERS-LEE, 2001; DICKINSON, 2003).

Outra nova área de aplicação para Agentes surgiu com o paradigma da Computação Orientada a Serviços (*SOC*). Já existem alguns trabalhos que buscam integrar propriedades e tecnologias de Agentes e *Web Services* (GIBBINS, 2003; TSAI, 2003; VAN DEN HEUVEL, 2003), o que tem relação com um ramo emergente da Engenharia de *Software*: a Engenharia de *Software* Orientada a Agentes.

2.4.5 Agentes versus Componentes de *Software*

Hoje, a abordagem predominante no mercado de desenvolvimento de *software* é a utilização de componentes.

Assim como Agentes, componentes não possuem uma definição amplamente aceita do termo. Uma definição mais global é a do *OMG*[25] (*Object Management Group*): segundo o grupo, um componente é um fragmento de *software* auto-contido, que possui uma interface bem definida (ou um conjunto de interfaces), que pode ser distribuído e instalado de forma independente, e que pode ser facilmente combinado e colaborar com outros componentes (AMOR, 2000; LIND, 2001; KRUTISCH, 2003).

O uso de tecnologias de *Web Services* é a forma mais nova de se trabalhar com componentes (LIND, 2001; YANG, 2003).

[25] Informações podem ser encontradas em: <http://www.omg.org/>

A questão principal é o foco de cada uma das abordagens: enquanto componentes visam aspectos como reusabilidade e customização do *software*, e são passivos, Agentes de *Software* estão centrados no processamento de tarefas complexas e agem de forma autônoma, como uma comunidade. Ambas abordagens atendem a pontos importantes e o que se tem feito é uni-las, buscando tirar proveito da combinação entre as habilidades de comunicação dos Agentes de *Software* e de reuso/parametrização dos componentes (AMOR, 2000; GARCIA, 2003; KRUTISCH, 2003).

Krutisch, Meyer e Wirsing (2003) apresentam dois diferentes conceitos segundo os quais o uso combinado de Agentes e Componentes de *Software* pode ser feito:

- "agentificação" – considera as tecnologias de componentes como o ponto de partida e tenta adicionar características de Agentes de *Software* aos componentes;
- "componentização" – considera as tecnologias de Agentes de *Software* como o ponto de partida e tenta adicionar características de componentes aos Agentes.

No entanto, ainda são necessários estudos para que se definam metodologias e se consolide o campo da Engenharia de *Software* Orientada a Agentes.

2.5 *WEB SERVICES*

Web Services podem ser vistos como Componentes de *Software*, independentes de implementação ou plataforma, que podem ser: descritos utilizando uma linguagem de descrição de serviços; publicados em um "diretório" de serviços; encontrados através de mecanismos de busca padronizados; que são invocados através de uma *API*, via rede; combinados com outros serviços (OELLERMANN, 2001; HANSEN, 2002, 2003; NEWCOMER, 2002).

Segundo definição do *W3C*, um *Web Service* é:

> (...) uma aplicação de *software* identificada por um *URI*[26], cujas interfaces e ligações são capazes de ser definidas, descritas e descobertas como artefatos *XML*. Um serviço *Web* suporta interações diretas com outros Agentes de *software* usando mensagens baseadas em *XML*, trocadas via protocolos baseados na *Internet* (W3C WEB SERVICES ARCHITECTURE WORKING GROUP, 2003).

De forma mais simplificada, pode-se dizer que um *Web Service* é um programa disponibilizado na *Web*, que pode ser utilizado por alguma aplicação que estejamos desenvolvendo. Para fazer a chamada do serviço, precisamos saber o endereço onde este programa está, os parâmetros de que ele necessita e o que ele pode retornar. Se já conhecemos essas informações, podemos escrever a chamada diretamente no código de nossa aplicação; caso contrário, podemos fazer com que nossa aplicação consulte uma base de serviços e descubra como utilizar o programa desejado.

Web Services combinam os melhores aspectos do desenvolvimento baseado em componentes na *Web*. Assim, como Componentes de *Software*, *Web Services* representam uma funcionalidade *black box* que pode ser reutilizada sem a preocupação com a linguagem e o ambiente utilizados em seu desenvolvimento (GRAHAM, 2002; HANSEN, 2003).

Devido aos benefícios originados da *XML*, *Web Services* propiciam ligação dinâmica de serviços e aumentam a interoperabilidade entre linguagens e plataformas (FERRIS, 2003).

Web Services são construídos para serem acessados por outras aplicações (OELLERMANN, 2001). Este acesso não é feito via protocolos específicos de modelos de objetos como *DCOM*, *RMI* e *IIOP*, mas sim via protocolos de transporte como *HTTP*, *FTP* e *SMTP* (HANSEN, 2002, 2003).

Uma forma mais primitiva de *Web Service*, apresentada por Newcomer (2002), é mostrada a seguir.

---

[26] Informações podem ser encontradas em: <http://www.w3.org/Addressing/>

```
http://internal.iona.com:8080/iona/phonelist.jsp?search=vinoski
```

Neste tipo de serviço, dados de entrada são passados como parâmetro no endereço da página *Web* que implementa o serviço. O exemplo é de uma função que retorna um telefone e endereço de *e-mail* a partir do nome fornecido.

Hoje, *Web Services* implicam no uso de uma arquitetura, padrões de tecnologias e diversos recursos para desenvolvimento.

2.5.1 Arquitetura de *Web Services*

A arquitetura de *Web Services* está baseada nas interações de três papéis: provedor, solicitante e registro de serviço (OELLERMANN, 2001; HANSEN, 2002, 2003; NEWCOMER, 2002; FERRIS, 2003). A Fig. 2.13 ilustra as interações entre esses papéis.

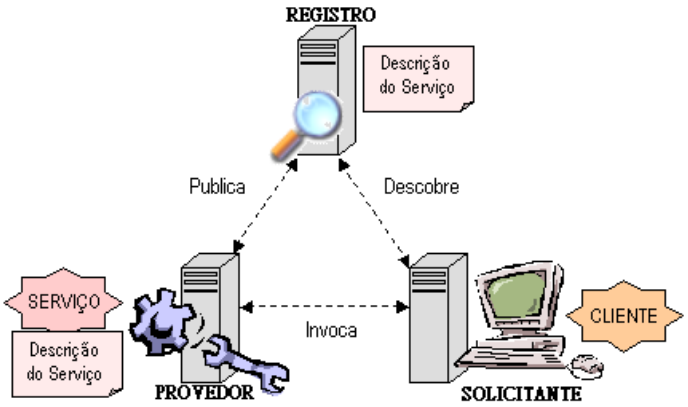


FIGURA 2.13 – Interações entre papéis na arquitetura de *Web Services*.

O provedor de serviço é a plataforma acessada na solicitação do serviço. É a entidade que cria o *Web Service*, sendo responsável por fazer sua descrição em algum formato padrão e publicar os detalhes em um registro de serviço central.

O registro de serviço é o local onde os provedores publicam as descrições dos serviços. Uma descrição de serviço torna possível descobrir onde está um *Web Service* e como

invocá-lo: o provedor cria uma descrição que detalha a interface do serviço, ou seja, suas operações e as mensagens de entrada e saída para cada operação; uma descrição de ligação é então criada, mostrando como enviar cada mensagem para o endereço onde o *Web Service* está localizado.

O solicitante de serviço é uma aplicação que invoca ou inicializa uma interação com um serviço. Pode ser um *browser* ou um programa sem interface com o usuário como, por exemplo, outro *Web Service*. Um solicitante de serviço encontra uma descrição de serviço, ou consulta o registro de serviço para o tipo de serviço requerido, e obtém as informações de ligação da descrição do serviço durante a fase de desenvolvimento (ligação estática) ou em tempo de execução (ligação dinâmica).

A execução das interações entre os papéis acontece via rede, suportada por um conjunto de tecnologias padronizadas. A Fig. 2.14 traduz esse cenário para um conjunto de camadas conceituais.

FIGURA 2.14 – Camadas conceituais de *Web Services*, adaptado de Hansen (2002, p. 3).

A rede é a camada base. Ela abrange os protocolos de transporte citados anteriormente (*HTTP*, *FTP*, *SMTP*) e pode ser utilizada para implementação de necessidades das aplicações, tais como: disponibilidade, performance, segurança e confiabilidade. Mensagem baseada em *XML* é a cama de comunicação. Ela utiliza o protocolo *SOAP* para realizar a troca de mensagens entre provedor, registro e solicitante. A descrição de serviços é feita com uma linguagem específica: a *WSDL*. As camadas de publicação e descoberta de serviços usam *UDDI* para tornar *Web Services* disponíveis (HANSEN, 2002, 2003; FERRIS, 2003).

2.5.2 Tecnologias

Por trás da estrutura básica de *Web Services*, existem algumas tecnologias padrão para sua construção: *Web Services Description Language* (*WSDL*), *Simple Object Access Protocol* (*SOAP*) e *Universal Description, Discovery and Integration* (*UDDI*). Estas tecnologias são baseadas em *XML* e permitem reutilização e chamada dos serviços sem que se conheça sua linguagem ou plataforma.

A Fig. 2.15 relaciona as tecnologias com os papéis previamente apresentados.

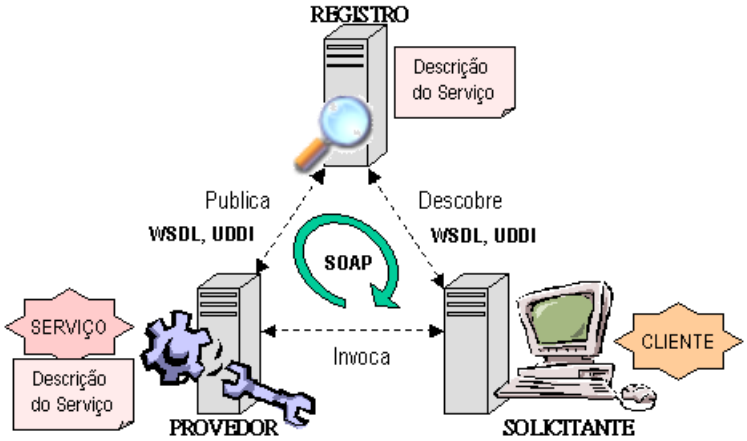


FIGURA 2.15 – Papéis e tecnologias relacionadas.

A linguagem *WSDL*[27] descreve um serviço como uma coleção de operações que podem ser acessadas através de mensagens. Utilizando seus métodos, é possível descrever um serviço de forma transparente e independente de implementação. As duas partes envolvidas em uma interação de *Web Services* precisam ter acesso à mesma descrição *WSDL* para conseguirem entender uma à outra (HANSEN, 2002, 2003; NEWCOMER, 2002).

A descrição de um serviço consiste de duas partes, mostradas na Fig. 2.16: definição da implementação do serviço e definição da interface do serviço.

[27] Informações podem ser encontradas em: <http://www.w3.org/TR/2004/WD-wsdl20-20040326/>

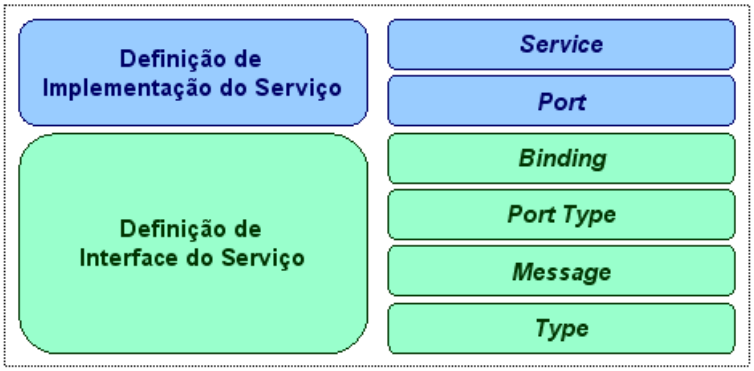


FIGURA 2.16 – Detalhamento da camada de descrição de serviços, adaptado de Hansen (2002, p. 3).

A *WSDL* também define protocolos de ligação e detalhes de rede. Ela apresenta descrições adicionais como contexto, *QoS* (*Quality of Service*) e relacionamento entre serviços. A separação entre as duas definições permite que elas sejam utilizadas separadamente (HANSEN, 2002, 2003).

A parte de definição de interface do serviço descreve o *Web Service*, incluindo métodos que são invocados e parâmetros que são enviados. Pode ser usada, instanciada e referenciada por múltiplas definições de implementação de um serviço, e consiste de algumas diretivas (HANSEN, 2002; NEWCOMER, 2002):

- `WSDL:binding` – descreve protocolos, formato de data, segurança e outros atributos para uma interface (*portType*) em particular;
- `WSDL:portType` – informa elementos de operações do *Web Service*;
- `WSDL:message` – define entrada e saída de dados referentes a operações. Pode assumir a forma de um documento inteiro ou de argumentos que devem ser mapeados para invocação de métodos;
- `WSDL:type` – define tipos de dados complexos em uma mensagem.

A parte de definição de implementação do serviço descreve como uma interface de serviço é implementada por um provedor: onde o serviço está instalado e como pode ser

acessado. A definição de um serviço (`WSDL:service`) contém uma coleção de elementos `WSDL:port` com um elemento `WSDL:binding`. Pode conter definições de extensibilidade. `WSDL:port` é a combinação de um `WSDL:binding` com um endereço de rede, fornecendo o endereço alvo para comunicação com o serviço (HANSEN, 2002; NEWCOMER, 2002).

O trecho a seguir define um exemplo[28] simplificado de uma descrição *WSDL*.

```
<message name="getTermRequest">
   <part name="term" type="xs:string"/>
</message>
<message name="getTermResponse">
   <part name="value" type="xs:string"/>
</message>
<portType name="glossaryTerms">
  <operation name="getTerm">
      <input message="getTermRequest"/>
      <output message="getTermResponse"/>
  </operation>
</portType>
```

Cada elemento `message` define as partes de uma mensagem e os tipos de dados associados, ou seja, define os elementos de dados de uma operação. No exemplo, temos a definição de uma mensagem nomeada `getTermRequest`, que possui um elemento chamado `term`, do tipo `string`. Temos, também, a definição de uma mensagem nomeada `getTermResponse`, que possui um elemento chamado `value`, do tipo `string`. Em comparação com a programação tradicional, é como se tivéssemos uma função `getTermRequest` com o parâmetro `term` e uma função `getTermResponse` com o parâmetro `value`.

O elemento `portType` define o *Web Service*, as operações que podem ser realizadas e as mensagens que estão envolvidas. No exemplo, `glossaryTerms` é definido como um elemento de operação do *Web Service*, onde `getTerm` é o nome de uma operação. A operação `getTerm` tem uma mensagem de entrada chamada `getTermRequest` e uma mensagem de saída chamada `getTermResponse` (definidas anteriormente no documento *XML*, através dos

[28] Exemplo obtido em: <http://www.w3schools.com/wsdl/wsdl_documents.asp>

elementos `message`). Fazendo uma comparação com a programação tradicional, `glossaryTerms` seria uma biblioteca de funções, e a operação `getTerm` seria uma função com `getTermRequest` como parâmetro de entrada e `getTermResponse` como parâmetro de retorno.

Conforme apresentado por Hansen (2002), a *WSDL* também especifica extensões relativas a protocolos e formatos de mensagem como *SOAP*, *HTTP* GET/POST e *Multipurpose Internet Mail Extensions* (*MIME*).

O protocolo *SOAP*[29] permite comunicação entre diversas aplicações em um ambiente distribuído e descentralizado. Como o formato das mensagens é baseado em *XML*, ele pode ser entendido por quase todas as plataformas de *hardware*, sistemas operacionais, linguagens de programação e equipamento de rede. Este protocolo é utilizado para publicar, localizar e invocar *Web Services*. Suporta *RPC* e pode trabalhar com protocolos como *HTTP* e *SMTP*. Ainda, possui definição de tipos de dados para as estruturas mais comumente utilizadas, como `string`, `integer`, `float`, `double` e `date` (HANSEN, 2002, 2003).

Segundo é apresentado por Hansen (2002, 2003), um pacote *SOAP* consiste de quatro partes:

- envelope – guarda o conteúdo da mensagem, quem pode processá-la e o quão obrigatório é fazer esse processamento. É uma estrutura que encapsula elementos sintáticos da mensagem;
- codificação – define mecanismos de serialização que podem ser utilizados para trocar instâncias ou tipos de dados definidos por uma aplicação;
- *RPC* – especifica como encapsular chamadas remotas de métodos e respostas dentro da mensagem;
- *Framework* de ligação e transporte – define um *Framework* abstrato para troca de envelopes *SOAP* entre aplicações utilizando um protocolo de transporte simples.

[29] Informações podem ser encontradas em: <http://www.w3.org/TR/soap/>

Há outros conceitos importantes apresentados por Seely, Tidwell e Kulchenko (2002), Hansen (2002, 2003) e Snell (2002). São eles:

- servidor *SOAP* – responsável por executar uma mensagem *SOAP*, agir como um interpretador e realizar trocas de mensagens;
- mensagem *SOAP* – consiste de um envelope, contendo cabeçalhos opcionais, e um corpo, contendo uma mensagem atual com seus parâmetros ou resultados. Esta estrutura e a navegação entre nodos são mostrados na Fig. 2.17.

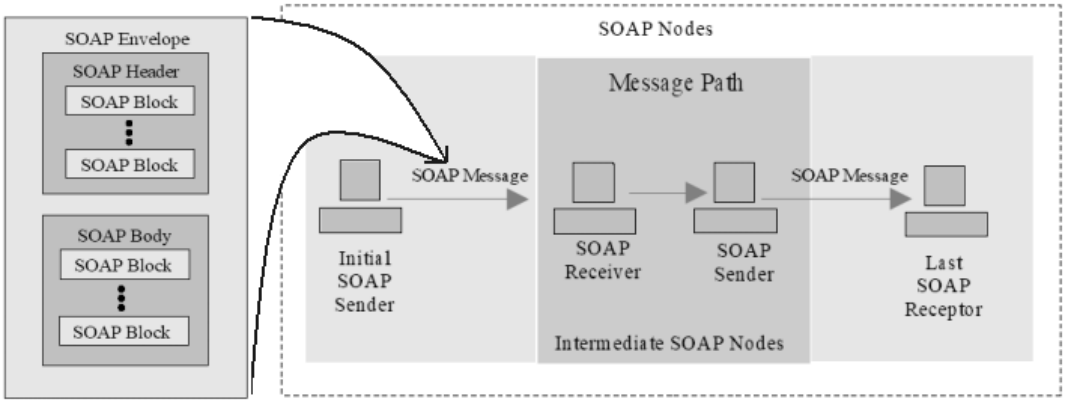


FIGURA 2.17 – Mensagem *SOAP* e a navegação entre nodos, adaptado de Hansen (2002, p. 5).

A mensagem *SOAP* constitui, logicamente, uma única unidade computacional. O bloco é identificado por um elemento externo chamado *namespace*[30]. O cabeçalho *SOAP* é uma coleção de zero ou mais blocos, os quais podem ser direcionados para um determinado receptor *SOAP* dentro do caminho da mensagem. O corpo *SOAP* é uma coleção de zero ou mais blocos direcionados para o último receptor *SOAP*. O protocolo *SOAP* não garante roteamento, apenas sabe qual o nodo que criou a mensagem e qual deve ser o último receptor da mesma, através de zero ou mais nodos intermediários. Quando um nodo recebe a mensagem, deve processá-la e gerar as mensagens adequadas – de sucesso, falha, ou outras adicionais (HANSEN, 2002, 2003; SEELY, 2002; SNELL, 2002).

[30] Informações podem ser encontradas em: <http://www.w3.org/TR/REC-xml-names/>

Os trechos a seguir mostram um pequeno exemplo[31] de código *SOAP*.

Requisição *SOAP* do exemplo:

```
POST /InStock HTTP/1.1
Host: www.stock.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 250
<?xml version="1.0"?>
<soap:Envelope
 xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
 soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.stock.org/stock">
    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Resposta *SOAP* do exemplo:

```
HTTP/1.1 200 OK
Content-Type: application/soap; charset=utf-8
Content-Length: 250
<?xml version="1.0"?>
<soap:Envelope
 xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
 soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.stock.org/stock">
    <m:GetStockPriceResponse>
      <m:Price>34.5</m:Price>
    </m:GetStockPriceResponse>
  </soap:Body>
</soap:Envelope>
```

Um cliente *HTTP* faz conexão a um servidor *HTTP* utilizando *TCP*. Após estabelecer uma conexão, o cliente pode enviar uma mensagem de requisição para o servidor. Uma requisição *SOAP* pode ser do tipo *HTTP* `POST` ou *HTTP* `GET`. A requisição *HTTP* `POST` especifica pelo menos dois cabeçalhos: `Content-Type` and `Content-Length`. O primeiro define o tipo *MIME* para a mensagem (`application/soap`, no exemplo) e a codificação de caracteres utilizada no corpo da requisição/resposta (`utf-8`); o segundo especifica o número de *bytes* no corpo da requisição/resposta (250 *bytes*). Após receber a requisição, o servidor

[31] Exemplo obtido em: <http://www.w3schools.com/soap/soap_example.asp>

processa a mesma e envia a resposta para o cliente. Essa resposta contém um código que indica o estado da requisição. No exemplo, o servidor retornou o código 200, que indica sucesso. Se o servidor não conseguisse decodificar a requisição, teria retornado:

```
HTTP/1.1 400 BAD REQUEST
Content-Length: 0
```

O envelope é o elemento raiz da mensagem, que define o documento *XML* como sendo uma mensagem *SOAP*. Este elemento, delimitado pelas *tags* `<soap:Envelope>` e `</soap:Envelope>`, utiliza dois *namespaces*, definidos em `xmlns:soap` e `soap:encodingStyle`. Os endereços definidos para esses *namespaces* são sempre `http://www.w3.org/2001/12/soap-envelope` e `http://www.w3.org/2001/12/soap-encoding`, respectivamente. O primeiro está relacionado ao próprio envelope *SOAP* e, o segundo, aos tipos de dados utilizados no documento *SOAP*. *Namespaces* são utilizados para evitar conflitos de documentos *XML* que apresentem os mesmos nomes para descrever tipos diferentes de elementos.

O corpo da mensagem, delimitado pelas tags `<soap:Body>` e `</soap:Body>`, define o conteúdo da mensagem *SOAP*. Nesta parte do documento *SOAP*, mais um *namespace* pode ser definido. Para isso, o endereço utilizado no exemplo é `http://www.stock.org/stock`.

No exemplo, uma requisição chamada `GetStockPrice` (solicitação do preço de um produto) é enviada para um servidor. A requisição tem o parâmetro `StockName` (referente ao nome do produto) definida no corpo da mensagem. A resposta a essa requisição, chamada `GetStockPriceResponse`, traz um parâmetro `Price` (o preço do produto consultado) definido no corpo da mensagem. Assim, é solicitado o preço de um produto identificado por "*IBM*" e o valor obtido como resposta corresponde a $34,50. Os elementos definidos dentro

do corpo da mensagem (`GetStockPrice`, `StockName`, `GetStockPriceResponse` e `Price`) são específicos da aplicação – não fazem parte do padrão *SOAP*.

Para invocar um serviço utilizando *SOAP*, uma aplicação requisita uma mensagem *SOAP* e invoca o serviço através de um provedor de *Web Services*. O solicitante apresenta a mensagem, que inclui o endereço do provedor de serviço na rede. A infraestrutura de rede então envia a mensagem para um servidor *SOAP*, que redireciona a mensagem para o provedor de serviços. O servidor *Web* do provedor é responsável por processar a mensagem de requisição e construir a resposta, que é redirecionada através da infraestrutura *SOAP*. Quando a mensagem *XML* chega ao solicitante, é convertida para uma linguagem de programação e enviada para a aplicação (HANSEN, 2002).

Para publicar ou encontrar um serviço, é necessário acessar um registro *UDDI*, que roda em um servidor.

A especificação *UDDI*[32] é um trabalho conjunto para criação de um registro de serviços padronizado. Ela possui um componente central chamado *UDDI Project*, que manipula um registro global e público chamado *business registry*. A informação oferecida pelo *business registry* consiste de três componentes (HANSEN, 2002, 2003; NEWCOMER, 2002):

- *white pages* – endereço, contato e identificadores conhecidos;
- *yellow pages* – categorização industrial;
- *green pages* – informação.

A implementação *UDDI* é um servidor de registro que fornece um mecanismo para publicar e localizar serviços. Guarda informações categorizadas sobre empresas, serviços que elas oferecem e a associação com as especificações desses serviços, feitas em *WSDL* através do próprio registro (HANSEN, 2002).

[32] Informações podem ser encontradas em: <http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm>

Registros podem ser públicos, acessados via *Internet*, ou privados, acessados em *Intranets* de empresas, por exemplo. Registros *UDDI* podem ser acessados tanto por aplicações, diretamente via código, quanto por pessoas, através de alguma interface. A Fig. 2.18 mostra a interface *Web* do servidor de registro[33] *UDDI* da *IBM*.

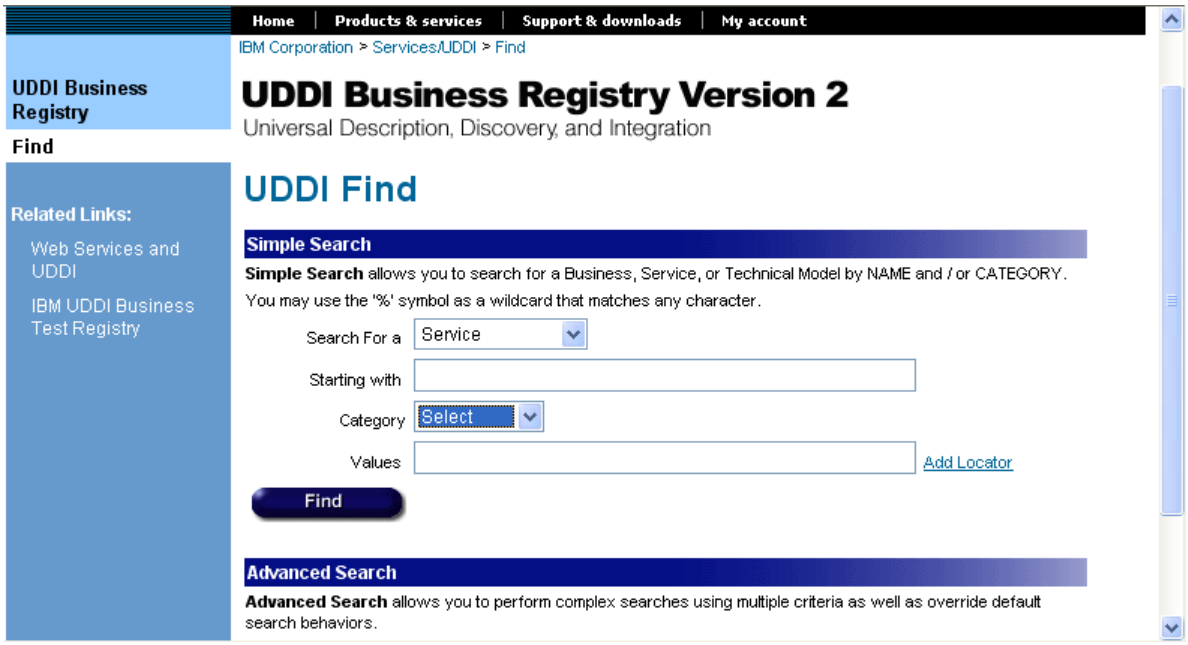


FIGURA 2.18 – Interface *Web* do servidor de registro *UDDI* da *IBM*.

As classificações para categorização das informações em um registro *UDDI*, apresentadas por Newcomer (2002), incluem:

- *North American Industry Classification System*[34] (*NAICS*);
- *Universal Standard Products and Services Classification*[35] (*UNSPSC*);
- *Internacional Organization for Standardization*[36] (*ISO*).

O modelo de informação principal utilizado pelo registro *UDDI* é definido através de *XML Schema*, englobando quatro tipos de informação: sobre o negócio, sobre o serviço, de ligação e específica do serviço (HANSEN, 2002, 2003).

[33] Informações podem ser encontradas em: <https://uddi.ibm.com/ubr/registry.html>
[34] Informações podem ser encontradas em: <http://www.census.gov/epcd/www/naics.html>
[35] Informações podem ser encontradas em: <http://www.unspsc.org/>
[36] Informações podem ser encontradas em: <http://www.iso.org/>

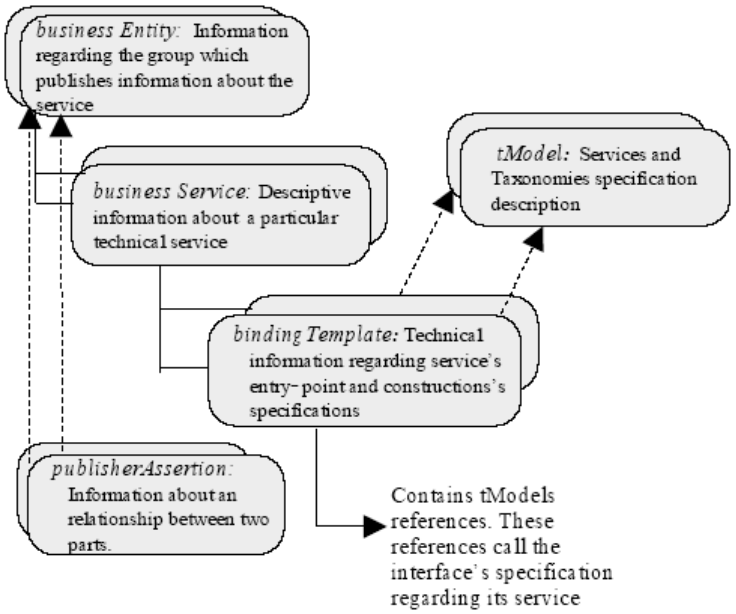


FIGURA 2.19 – Estrutura *UDDI* (HANSEN, 2002, p. 6).

Alguns tipos de estruturas de dados, mostradas na Fig. 2.19, também compõem o registro (HANSEN, 2002, 2003; NEWCOMER, 2002). São elas:

- `businessEntity` – estrutura de alto nível que representa toda a informação conhecida sobre uma empresa específica ou entidade que publica informação descritiva, bem como serviços;
- `businessService` – representa uma classificação lógica do serviço. Cada estrutura `businessService` pertence a uma única estrutura `businessEntity`;
- `bindingTemplate` – estruturas deste tipo são descrições técnicas sobre *Web Services* registrados. Fornecem suporte para que se possa acessar os serviços remotamente, e descrevem como a estrutura `businessService` utiliza várias informações técnicas;
- `tModel` – é representado através de metadados e tem por objetivo fornecer um sistema de referência;
- `publisherAssertion` – permite que se possa associar estruturas *businessEntity*, de forma a obter uma melhor identificação. Cada uma das partes relacionadas

precisa concordar que o relacionamento entre elas é válido e precisam publicar exatamente a mesma informação, mantendo seu relacionamento visível.

As informações contidas em arquivos de descrição de serviço (*WSDL*) complementam aquelas que estão no registro. No entanto, *UDDI* fornece suporte a vários tipos de descrição de serviço, mas não suporta a criação de descrições *WSDL* de forma direta.

Uma descrição *WSDL* completa consiste da combinação dos documentos de interface e de implementação do serviço. A descrição de implementação é publicada no registro *UDDI* como um `businessService` ou dentro de um `bindingTemplate` (ou mais de um). A interface pode ser publicada no registro usando um *`tModel`*, o que deve ser feito antes de a implementação ser publicada como `businessService` (HANSEN, 2002, 2003).

Hoje existem diversos recursos que facilitam o desenvolvimento de aplicações *Web Services*, e esta está se tornando uma área de crescente pesquisa.

2.5.3 Recursos para o desenvolvimento de *Web Services*

Devido à interoperabilidade proporcionada pelos padrões *Web Services*, suas tecnologias estão sendo cada vez mais utilizadas por organizações, visando tanto a integração de sistemas em *Intranet* como a disponibilização de serviços de negócio via *Web* e integração de sistemas de diferentes negócios também utilizando a *Internet* (OGBUJI, 2002b; RIST, 2002; CHUNG, 2003; FOSTER, 2003; GEER, 2003).

Assim, muitos trabalhos vêm sendo desenvolvidos com o objetivo de melhorar a capacidade de integração dos *Web Services*.

Algumas soluções aplicam Ontologias, Agentes, linguagens e ferramentas de *Web* Semântica e outros recursos de Inteligência Artificial (BUSSLER, 2002; MEDJAHED, 2003; PAHL, 2003; TSAI, 2003; VAN DEN HEUVEL, 2003; WILLIAMS, 2003). Outras visam o

controle dos serviços disponibilizados, visando gerenciar as formas de uso, o desempenho e outros fatores, de acordo com os objetivos de negócio (CASATI, 2003; CURBERA, 2003; PELTZ, 2003; TERAI, 2003). A segurança, um ponto ainda fraco, tem sido foco de outros trabalhos (DAMIANI, 2002; GORDON, 2002; KRAFT, 2002; GEER, 2003; NAEDELE, 2003; WEIPPL, 2003).

Várias empresas têm disponibilizado ferramentas e plataformas para auxiliar no desenvolvimento de *Web Services*, ou mesmo adicionado funcionalidades de *Web Services* em ferramentas já existentes. As principais plataformas disponíveis no mercado são:

- *Sun One*, da *Sun Microsystems*[37];
- *Microsoft .Net*[38];
- *IBM WebSphere*[39];
- *HP Web Services Platform*[40].

Uma descrição do conjunto destas plataformas pode ser encontrada em trabalhos de Hansen (2002, 2003).

Todas as plataformas citadas são proprietárias e seu uso implica no investimento de recursos financeiros. A fim de criar uma solução que não gerasse custos para poder ser utilizada, optamos por utilizar o pacote *JWSDP*[41] disponibilizado pela *Sun Microsytems*. A versão atual deste pacote consiste de:

- *JavaServer Faces* – tecnologia que simplifica a criação de interfaces para o usuário para aplicações *JavaServer* (por exemplo, páginas *JSP*);
- *XML and Web Services Security* – tecnologia que permite assinar e verificar mensagens *SOAP*;

[37] Informações podem ser encontradas em: <http://www.sun.com/software/products/dev_platform/home_devplat.html>
[38] Informações podem ser encontradas em: <http://www.microsoft.com/net/>
[39] Informações podem ser encontradas em: <http://www-306.ibm.com/software/info1/websphere/index.jsp>
[40] Informações podem ser encontradas em: <http://h21022.www2.hp.com/*HP*ISAPI.dll/hpmiddleware/products/hp_web_services/default.jsp>
[41] Informações podem ser encontradas em: <http://java.sun.com/webservices/webservicespack.html>

- *Java Architecture for XML Binding* (*JAXB*) – tecnologia que permite associar um *XML Schema* a código *Java*;
- *Java API for XML Processing* (*JAXP*) – tecnologia que permite processar e transformar documentos *XML*;
- *Java API for XML Registries* (*JAXR*) – tecnologia que permite a construção de aplicações que acessam diversos tipos de registros *XML* (por exemplo, registros *UDDI*);
- *Java API for XML-based RPC* (*JAX-RPC*) – tecnologia que permite a incorporação de funcionalidades *RPC* baseadas em *XML*, de acordo com a especificação *SOAP* 1.1;
- *SOAP with Attachments API for Java* (*SAAJ*) – tecnologia que oferece uma maneira padronizada de transferir documentos *XML* através da rede;
- *JavaServer Pages Standard Tag Library* (*JSTL*) – tecnologia que encapsula funcionalidades comuns à maioria das aplicações *Web* no formato de *tags* – por exemplo, uso de *Structured Query Language* (*SQL*);
- *Java* WSDP *Registry Server* – implementa a versão 2 da especificação *UDDI*, fornecendo um registro *UDDI* para uso em ambiente privado.

O pacote *JWSDP* vem acompanhado de uma versão do servidor *Web Apache Tomcat*[42].

Existem ainda diversas ferramenta gratuitas que podem ser encontradas na *Internet* como, por exemplo, aquelas disponibilizadas pela *Apache*[43].

2.6 APLICAÇÃO DAS TECNOLOGIAS ESTUDADAS

O *WSAgent* é uma solução que visa integrar bases de dados heterogêneas, buscando obter o máximo possível em interoperabilidade, flexibilidade e reuso.

[42] Informações podem ser encontradas em: <http://jakarta.apache.org/tomcat/index.html>
[43] Informações podem ser encontradas em: <http://apache.org>

A solução proposta consiste de *Web Services* no papel de Agentes de *Software*, trabalhando em pares, ambientados na *Intranet* de um hospital (podendo estar em diferentes máquinas, servidores de aplicação e plataformas) e comunicando-se via rede. Segundo os conceitos apresentados por Krutisch, Meyer e Wirsing (2003), optou-se por "agentificar" *Web Services*. As características incorporadas serão apresentadas no capítulo 4, onde o *WSAgent* é descrito.

Os *Web Services* farão uso de Ontologias para manter a compatibilidade entre as informações trocadas, e *XML* constitui a base de sua comunicação. O estudo realizado, incluindo a análise de Ontologias já definidas, serviu como base para o desenvolvimento das Ontologias que serão utilizadas na solução proposta. A necessidade de que os *Web Services* trabalhem com fontes de dados específicas no processo de integração de bases heterogêneas levou à definição de Ontologias próprias, adaptando esse conceito às necessidades da aplicação.

A camada de persistência do *WSAgent* faz uso o de pequenos *Frameworks*, chamados *Framelets* (PREE, 1999b, 2000), e *Design Patterns* para tornar flexível a customização da base de dados que um Agente deverá manipular.

O capítulo a seguir apresenta alguns projetos desenvolvidos para a integração de sistemas no domínio da saúde.

3 TRABALHOS RELACIONADOS

O mercado, na área da saúde, tem sofrido diversas mudanças nos últimos anos. Assim, tem se tornado cada vez maior a demanda por soluções que permitam a troca e a integração de informações entre fontes heterogêneas (FAYAD, 1999c; BIRD, 2000, 2002; KIM, 2001; KUHN, 2001).

Este capítulo apresenta os principais *Frameworks* que emergiram, neste contexto, nos últimos anos. Cada um enfoca uma camada específica da arquitetura de *software* (FAYAD, 1999c).

3.1 *HEALTH LEVEL SEVEN* (*HL7*)

Este *Framework* é o principal no mercado de integração de dados clínicos. Ele fornece suporte para que sistemas trabalhem juntos através da troca de mensagens na parte do servidor de aplicações, enfocando a troca de dados entre sistemas hospitalares. A integração através de mensagens produz uma camada de processos (definidos e gerenciados de forma centralizada) no topo dos processos existentes. O objetivo é combinar processos relevantes, de forma a dar suporte ao fluxo de informações e lógica de controle entre eles (FAYAD, 1999c; KUHN, 2001; MYKKÄNEN, 2004).

O *Framework HL7*[44] é desenvolvido por um conjunto de membros – fornecedores, vendedores, compradores, consultores, grupos governamentais e demais interessados –

[44] Informações podem ser encontradas em: <http://www.hl7.org>

conhecidos como *HL7 Working Group*. Este grupo de trabalho é organizado em comitês técnicos (diretamente responsáveis pelo desenvolvimento de padrões) e grupos de interesses especiais (responsáveis por explorar novas áreas que devem ser agregadas aos padrões). O grupo não desenvolve *software*, mas sim especificações que constituem padrões para troca de dados entre sistemas heterogêneos. Esta troca pode ser feita com base em tecnologias como *CORBA* e *ActiveX*[45], por exemplo (WIRSZ, 2000; HEALTH LEVEL SEVEN, 2004a).

O grupo, fundado em 1987 e sem fins lucrativos, é uma organização desenvolvedora de padrões reconhecida pela *ANSI*[46], tendo como domínio o que se refere a dados clínicos e administrativos. Sua missão é: "fornecer padrões para a troca, gerência e integração de dados que suportem cuidados clínicos ao paciente e o gerenciamento, distribuição e avaliação de serviços de saúde. Especificamente, criar abordagens, padrões, guias, metodologias e serviços relacionados para promover interoperabilidade entre sistemas de informação na área da saúde" (WIRSZ, 2000; HEALTH LEVEL SEVEN, 2004a).

A versão atual do conjunto de padrões para troca de mensagens é chamada *HL7 V3*. Esses padrões são desenvolvidos segundo uma metodologia OO e definem como informações clínicas são trocadas entre sistemas da área da saúde. *HL7 V3* engloba especificações de tipos abstratos de dados, *Reference Information Model* (*RIM*), *XML Implementation Technology Specification* (*XMLITS*) e vocabulário de domínio (FAYAD, 1999c; WIRSZ, 2000; HOODA, 2004).

A Fig. 3.1 mostra a estrutura do *RIM*, que é o ponto principal no desenvolvimento do *HL7*. Esse modelo representa os domínios de dados clínicos e identifica o ciclo de vida de eventos para uma mensagem ou um grupo de mensagens (LAROIA, 2002; HEALTH LEVEL SEVEN, 2004b; HOODA, 2004).

[45] Informações podem ser encontradas em: <http://www.microsoft.com/com/tech/ActiveX.asp>
[46] Informações podem ser encontradas em: <http://www.ansi.org/>

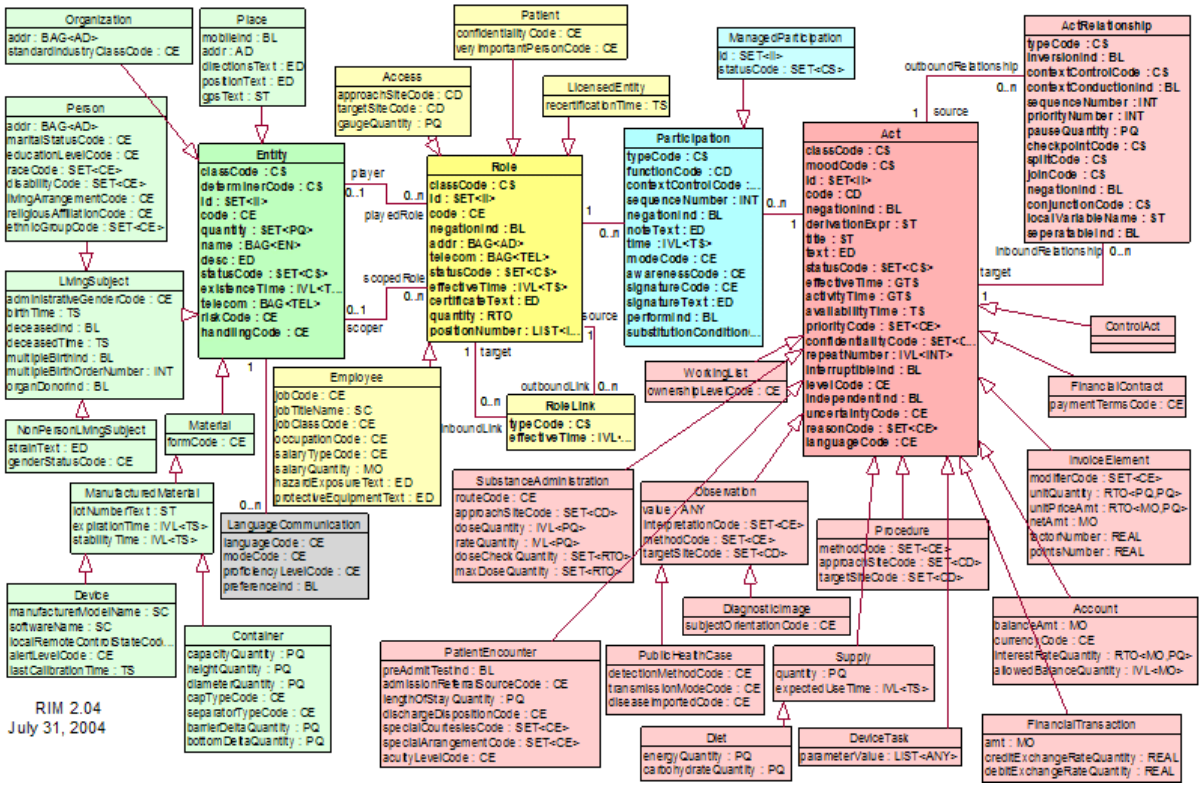


FIGURA 3.1 – Estrutura do *RIM* (HEALTH LEVEL SEVEN, 2004b)[47].

Há duas áreas principais no *RIM*: a infra-estrutura normativa e a infra-estrutura de comunicação. Enquanto a primeira descreve a gramática do *HL7 V3*, a segunda define formatos de mensagens e documentos, que devem permitir que aplicações da área da saúde se comuniquem umas com as outras (HOODA, 2004).

O *RIM* consiste de um conjunto de classes (representadas em *UML*), cada uma contendo um ou mais atributos que são associados a tipos referentes a uma especificação independente de tipos de dados (HEALTH LEVEL SEVEN, 2004b).

As classes que formam a espinha dorsal do modelo são:

- `Act` – representa as ações que são executadas e precisam ser documentadas à medida que o atendimento de saúde é gerenciado e fornecido;
- `Entity` – representa os seres e coisas físicas que são de interesse e tomam parte no atendimento de saúde;

[47] Uma versão mais legível da figura pode ser encontrada em: <http://www.hl7.org/library/data-model/RIM/C30204/rim.htm>

- `Participation` – expressa o contexto para uma ação em termos de quem a realizou, para quem foi realizada, onde foi realizada, etc.;
- `Role` – estabelece os papéis que entidades assumem enquanto participam de ações;
- `ActRelationship` – representa a ligação entre uma ação e outra, como o relacionamento entre uma ordem de observação e o evento de observação ocorrido;
- `RoleLink` – representa relacionamentos entre papéis individuais.

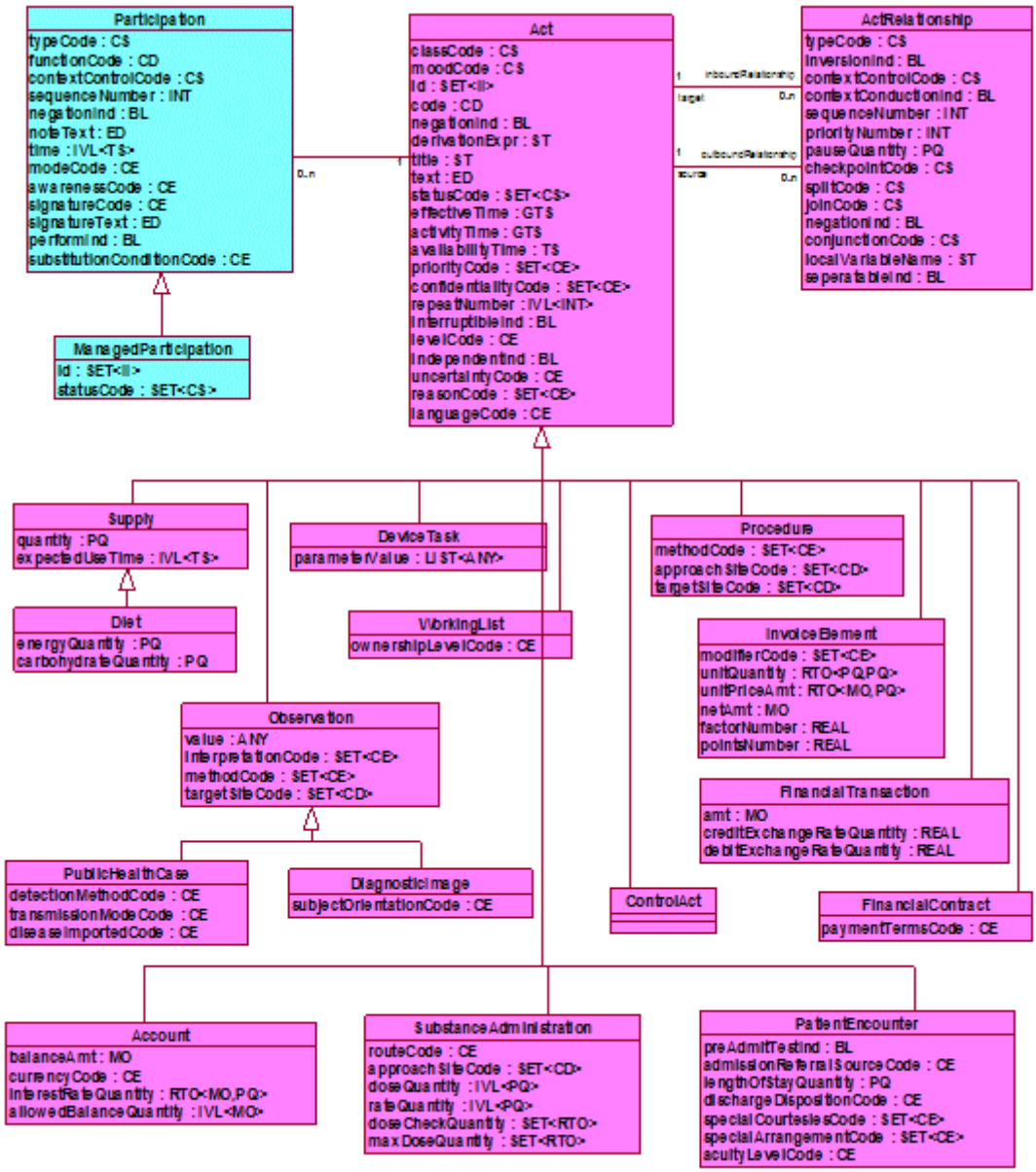


FIGURA 3.2 – Contexto da classe `Act` (HEALTH LEVEL SEVEN, 2004b)[48].

[48] Uma versão mais legível da figura pode ser encontrada em: <http://www.hl7.org/library/data-model/RIM/C30204/rim.htm>

A Fig. 3.2 mostra o contexto da classe `Act` e, a Fig. 3.3, seu diagrama de estados.

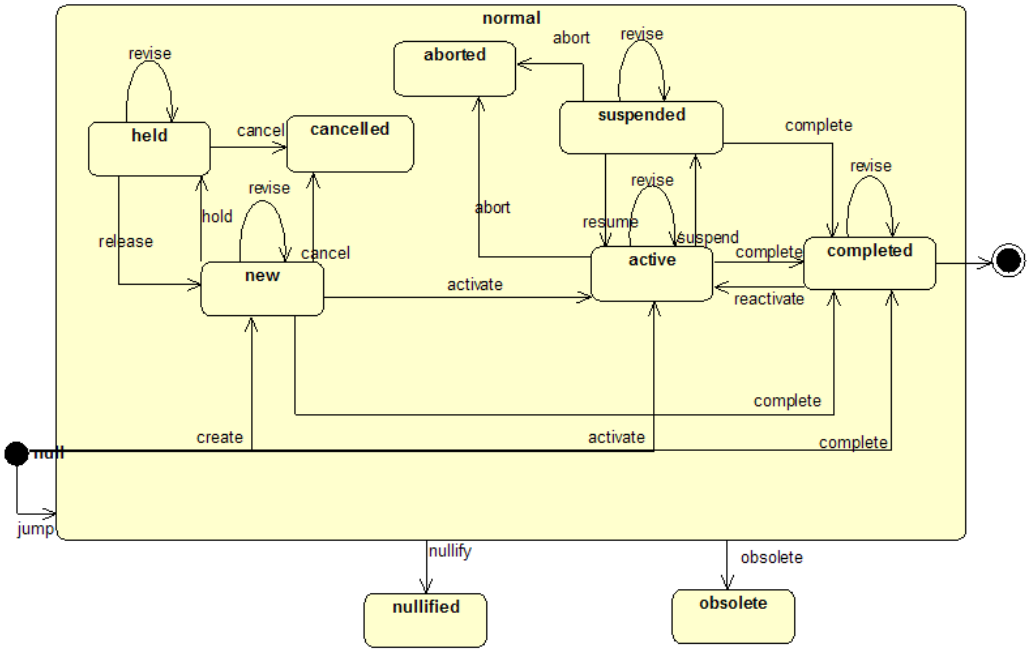


FIGURA 3.3 – Diagrama de estados da classe `Act` (HEALTH LEVEL SEVEN, 2004b).

Dentro da infra-estrutura de comunicação do *RIM*, dois componentes principais são identificados e construídos utilizando suas classes: *Clinical Document Architecture* (*CDA*) e *Message Development Framework* (*MDF*). Com o uso da *XML*, do *RIM* e de vocabulários codificados, a *CDA* permite que documentos sejam compartilhados entre diferentes sistemas. Os documentos são processados eletronicamente, de forma a permitir que sejam facilmente obtidos e utilizados pelos profissionais que deles necessitam. O *MDF* especifica a estrutura de mensagem de vários sistemas que trocam dados entre si. De forma geral, as mensagens podem incluir consultas, ordens, resultados, registros médicos, entre outros (LAROIA, 2002; HOODA, 2004).

O conjunto de padrões *HL7* não especifica a arquitetura de armazenamento e gerenciamento dos dados nas aplicações. Ele é projetado para trabalhar com sistemas onde os dados podem residir em uma base de dados central ou um conjunto de bases de dados locais, e atua como a espinha dorsal para que aplicações e arquiteturas de dados heterogêneas possam

se comunicar umas com as outras. Resultados de uma mensagem *HL7* são enviados a um repositório (banco de dados), onde podem ser obtidas por diferentes sistemas, para diferentes propósitos. Adicionar um novo sistema nesse ambiente requer a implementação de funcionalidades que manipulem os dados replicados – por exemplo, que permita visualizar esses dados (KUHN, 2001; HOODA, 2004).

A *XMLITS* é utilizada no desenvolvimento de *XML Schemas* para as especificações do *HL7 V3*. No entanto, este ainda é um ponto com diversas inconsistências, e que precisa ser aperfeiçoado (HOODA, 2004).

Em meados de maio de 2004, foi anunciada a aprovação do *Web Services Profile* e da *ebXML Message Service Specification* 2.0, na qualidade de *Draft Standards for Trial Use* (*DSTUs*). Essas especificações também fazem parte do *HL7 V3*, e foram introduzidas "em resposta a uma necessidade da indústria por um aumento na interoperabilidade entre implementações" (ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS, 2004).

O *HL7 Working Group* conta com diversos Comitês Técnicos[49] e Grupos de Interesse[50], onde são tratadas questões relativas a segurança, aplicação de Ontologias, entre outros.

3.2 *CORBAMED*

O *OMG* é o responsável pelo desenvolvimento desse *Framework*. O grupo, sem fins lucrativos, estabeleceu uma força tarefa específica para a área da saúde. Seus integrantes desenvolvem especificações baseadas em tecnologia *CORBA* – mais especificamente, serviços (ELLINGSEN, 2001; MYKKÄNEN, 2004).

[49] Informações podem ser encontradas em: <http://www.hl7.org/Special/committees/tc1.htm>
[50] Informações podem ser encontradas em: <http://www.hl7.org/Special/committees/hl7sigs.htm>

A integração orientada a serviços permite que aplicações compartilhem lógica de negócio ou métodos em comum. Isso é feito por meio da definição de métodos compartilhados e do fornecimento da infra-estrutura ou *middleware* para compartilhamento. Um conjunto de métodos em comum reduz a necessidade de replicar esses métodos (e os dados envolvidos) em diversas aplicações, e possibilita a integração orientada a informação/ processo por fornecer a infra-estrutura necessária (ELLINGSEN, 2001; MYKKÄNEN, 2004).

Alguns objetivos do desenvolvimento do conjunto de especificações *CORBAmed*, apresentadas por Ellingsen (2001) e pelo *Object Management Group* (2004a), são:

- Promover a interoperabilidade entre dispositivos e sistemas da área da saúde utilizando tecnologia *CORBA*;
- Promover a qualidade do atendimento e reduzir custos através do uso da tecnologia *CORBA*;
- Propiciar a troca segura e confiável de informações entre organizações da área da saúde.

O *CORBAmed* utiliza a *Object Management Architecture* (*OMA*), ilustrada pela Fig. 3.4.

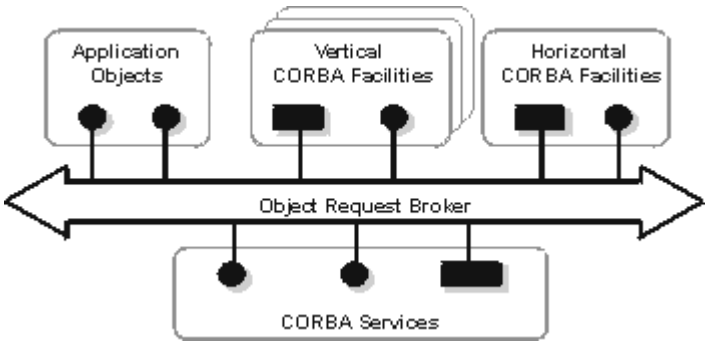


FIGURA 3.4 – Arquitetura *OMA* (OBJECT MANAGEMENT GROUP, 2004b).

Aplicações, mesmo quando realizam tarefas totalmente diferentes, compartilham uma série de funcionalidades em comum: objetos notificam outros objetos quando algo acontece; instâncias de objetos são criadas e compartilhadas; instâncias de objetos são criadas e destruídas. A *OMA* abstrai essas funcionalidades de aplicações *CORBA*, classificando-as segundo um conjunto de objetos que realizam funções bem definidas e padronizadas, e que

podem ser acessadas através de interfaces *OMG IDL*. Essas interfaces especificam o que os objetos fazem. Assim, empresas que desenvolvem *software* podem implementar e vender as implementações de objetos que realizam serviços específicos (OBJECT MANAGEMENT GROUP, 2004b).

A classificação de objetos na arquitetura *OMA*, segundo o *Object Management Group* (2004b), é feita segundo quatro categorias:

- *CORBAservices* – provê funcionalidades básicas realizando, para a aplicação, o tipo de serviço que bibliotecas de sistema realizam em sistemas tais como *UNIX*. Funcionalidades relativas a segurança e transações também fazem parte desta categoria;
- *CORBAfacilities* – funcionalidades úteis para o contexto de negócio (como impressão, internacionalização, etc);
- *CORBAdomain* – utiliza a *IDL* na definição de interfaces padrão para objetos que qualquer organização pode compartilhar;
- *Application* – refere-se a objetos que não são afetados por esforços de padronização do *OMG*.

Exemplos de serviços[51] já desenvolvidos para este *Framework*, conforme informado por Wirsz (2000), são:

- *Person Identification Service* (*PIS*) – provê as funcionalidades para identificação única de paciente, gerenciamento de IDs (criação, fusão, inativação) e co-relacionamento de múltiplos IDs;
- *Clinical Observation Access Service* (*COAS*) – prove as funcionalidades para envio e recebimento de observações clínicas, seleção de observações (por tipo, paciente, etc), consulta de observações e outros;

[51] Informações podem ser obtidas em: <http://healthcare.omg.org/Roadmap/corbamed_roadmap.htm>

- *Clinical Image Access Service* (*CIAS*) – provê as funcionalidades para acessar imagens médicas e dados relacionados de forma útil para os profissionais de saúde.

Nas transmissões, não são enviados apenas dados, mas também podem estar incluídas informações de segurança, serviços de validação de usuário, identificação de paciente, e outras que a tecnologia *CORBA* suporta (ELLINGSEN, 2001).

3.3 *FRAMEWORK* DE INTEGRAÇÃO VISUAL

O *Clinical Context Object Workgroup*[52] (*CCOW*) do *HL7* publica padrões para integração visual de aplicações no domínio da saúde, em computadores pessoais ou estações de trabalho. A integração é realizada através da informação que é apresentada para ou fornecida pelo cliente – identidade de um paciente, por exemplo. A integração orientada ao usuário permite que o mesmo obtenha uma visão consistente de diversos sistemas, o que pode ser realizado com a utilização de um sistema unificador ou através da sincronização de várias aplicações na estação de trabalho do usuário. Esta abordagem tem foco em aspectos do sistema que são individuais a cada usuário, e as aplicações não podem ser diretamente integradas em nível de dados ou serviços (FAYAD, 1999c; MYKKÄNEN, 2004).

O grupo publica padrões para integração visual com interação cooperativa entre aplicações de saúde. Quando, em uma aplicação, um paciente diferente é selecionado, todas as aplicações são notificadas com as chaves de identificação associadas com este paciente (por exemplo, o número de registro do paciente no hospital, o número do paciente na clínica, etc). O foco é a colaboração entre aplicações com interface gráfica em uma estação de trabalho: o gerenciador faz com que diferentes aplicações sejam unificadas, e assim cada aplicação individual refere-se ao mesmo paciente ou usuário (WIRSZ, 2000).

[52] Informações podem ser obtidas em: <http://www.ccow-info.com/>

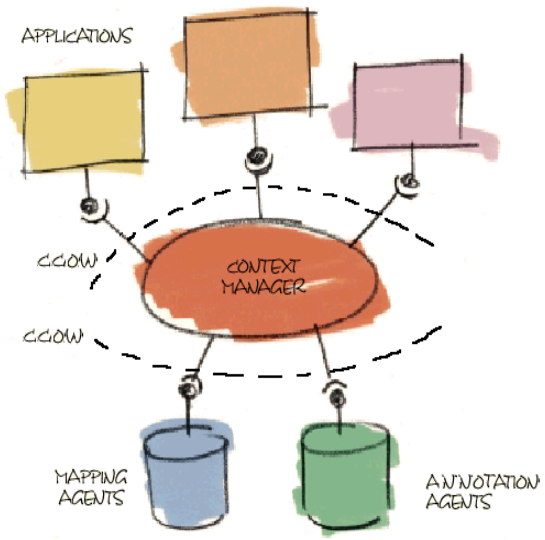


FIGURA 3.5 – Arquitetura *CCOW* (SELIGER, 2001).

A arquitetura utilizada, mostrada na Fig. 3.5, é composta de três tipos principais de componentes: aplicações; um gerenciador de contexto que coordena e sincroniza aplicações; e agentes de mapeamento que podem representar os vários sinônimos utilizados para identificar pacientes, usuários, etc. A arquitetura define papéis e responsabilidades para cada um desses componentes e prescreve precisamente as interfaces que habilitam a comunicação entre eles. A arquitetura não define ou impõe a implementação de nenhum dos componentes, e foi projetada para ser implementada em todos os tipos de sistemas da área da saúde, podendo utilizar diversas tecnologias (SELIGER, 2001).

Há ligações de contexto que podem ser gerais ou seguras. Qualquer aplicação pode obter ou definir dados de contexto para uma ligação geral. Em contraste, somente aplicações “nomeadas” podem acessar dados de contexto para uma ligação segura. As aplicações, o gerenciador de contexto e os agentes de mapeamento utilizam assinatura digital para autenticar as mensagens que eles enviam e recebem. A idéia básica é fazer com que os vários componentes *CCOW* possam confiar uns nos outros; por exemplo: fazer com que aplicações possam saber que estão se comunicando com o verdadeiro gerenciador de contexto e não com uma aplicação que está se fazendo passar pelo gerenciador (SELIGER, 2001).

O *Framework* especificado pelo grupo pode ser implementado em ambientes baseados, por exemplo, em *CORBA*, *COM* ou *Java* (FAYAD, 1999c).

A Fig. 3.6, a Fig. 3.7 e a Fig. 3.8 oferecem uma idéia de ambiente desenvolvido pelo grupo, o *Concerto Medical Applications Portal*. As imagens foram retiradas de uma demonstração[53] disponibilizada no *site* do grupo.

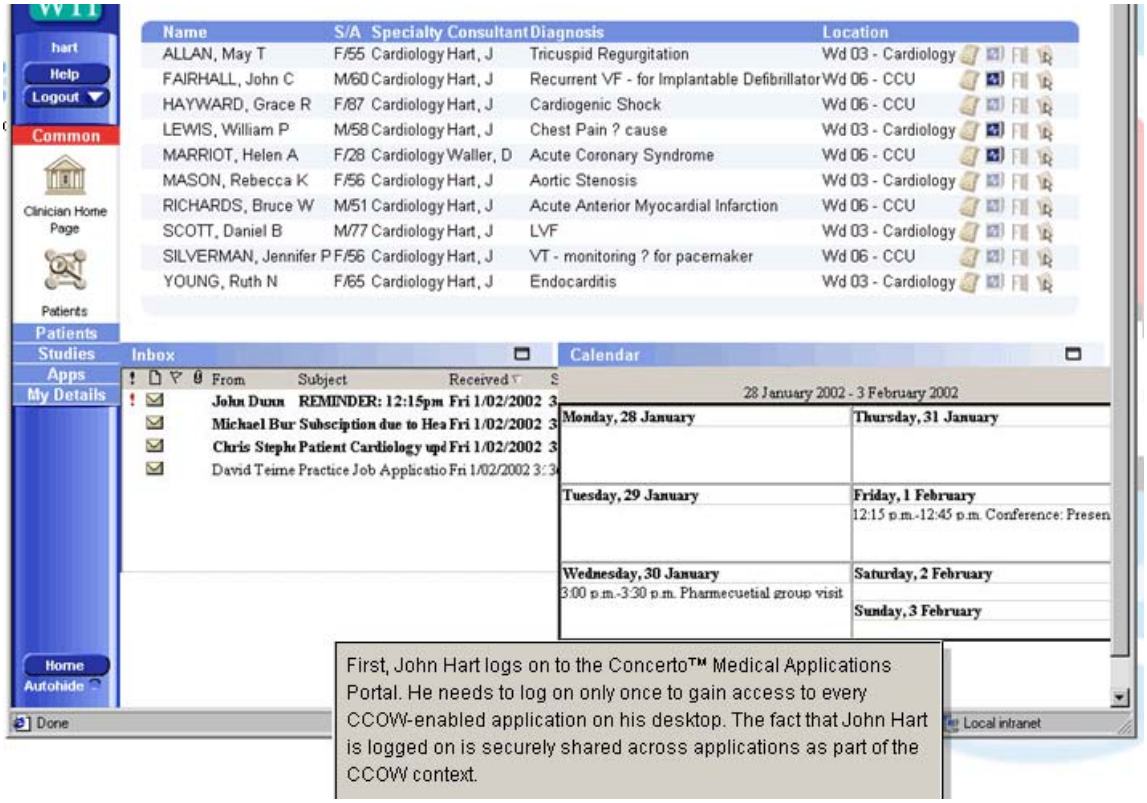


FIGURA 3.6 – *Concerto Medical Applications Portal*, mostrando lista de tarefas.

[53] O arquivo pode ser obtido em: <http://www.ccow-info.com/images/CCOWDemonstration.zip>

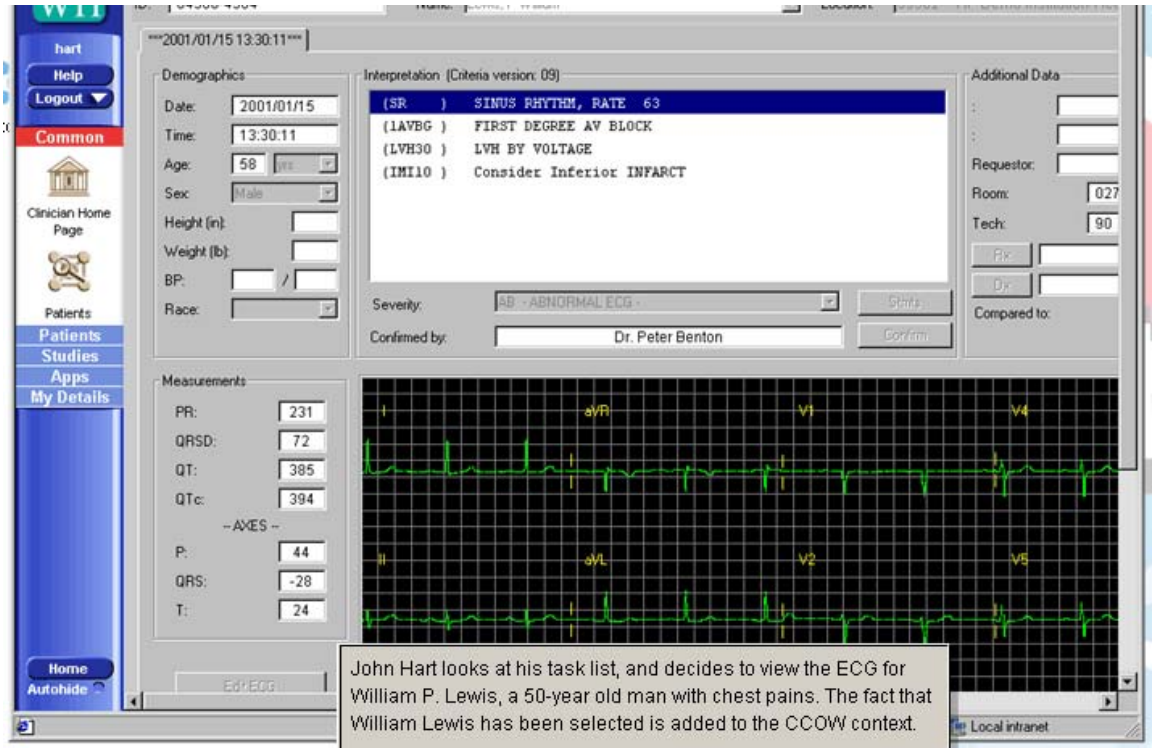


FIGURA 3.7 – Visualização do exame de um paciente.

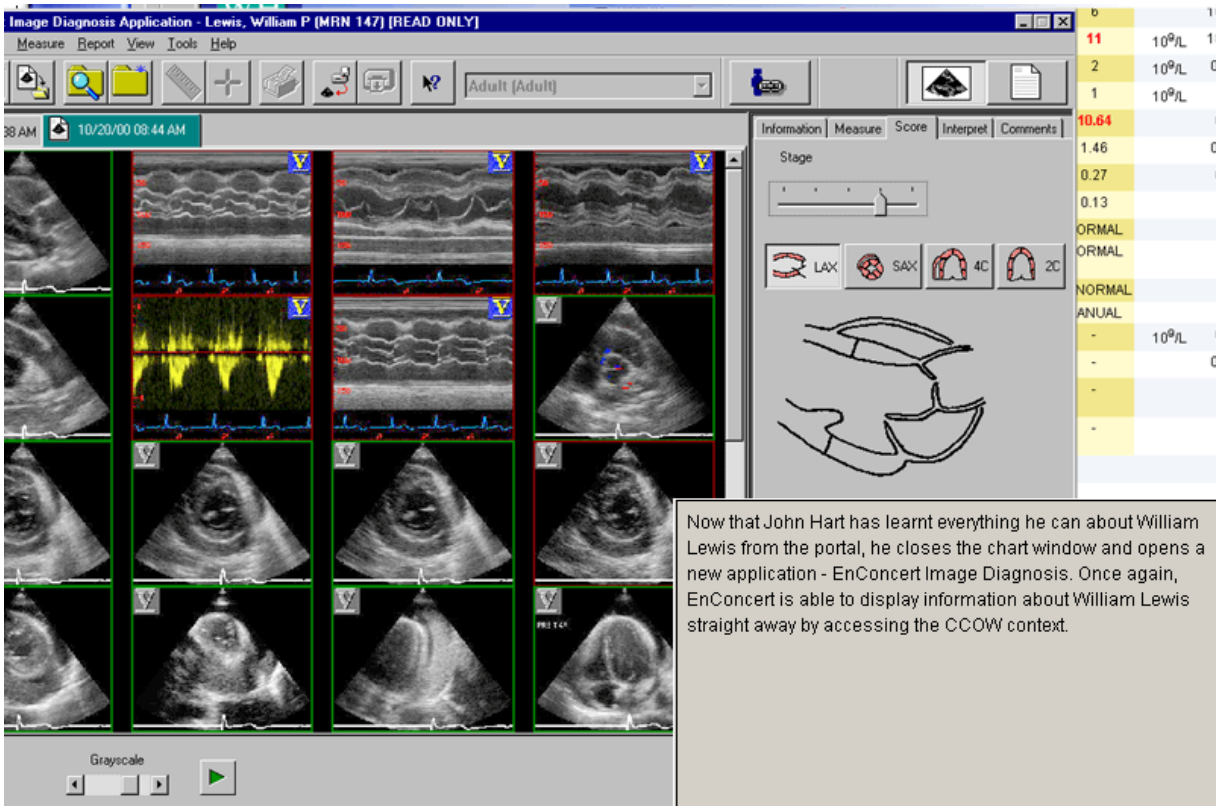


FIGURA 3.8 – Outra aplicação no contexto: *EnConcert Image Diagnosis*.

3.4 PROJETOS APRESENTADOS *VERSUS* SOLUÇÃO PROPOSTA

As soluções apresentadas neste capítulo visam integrar sistemas no domínio da saúde. Porém, os grupos que as desenvolvem estão mais concentrados na definição de padrões/ especificações.

Uma das principais características do *WSAgent* é a aplicação de tecnologias de *Web Services* – em sua maioria, padronizadas pela *W3C*. Dentre os trabalhos citados, apenas o *HL7* faz um grande uso de *XML*, que é base das tecnologias de *Web Services* (mais recentemente, outros recursos de *Web Services* foram incorporados para testes). Mas, como o *WSAgent* implementa serviços, essa característica se torna uma vantagem pelo fato de permitir a total distribuição da solução (instalação de serviços em diversos servidores de aplicação e diferentes máquinas).

O uso de *XML* e outros padrões derivados leva à promoção da interoperabilidade. Isso é válido para os *Frameworks WSAgent* e *HL7*. Já o *Framework CORBAmed* promove interoperabilidade através da aplicação de tecnologia *CORBA*.

Outra característica do *WSAgent* é a aplicação de Ontologias como um meio de compatibilização dos dados importados. Dos demais trabalhos, apenas o *HL7* explicita o uso de Ontologias para compatibilizar a troca de documentos clínicos entre sistemas.

Uma forte diferença ente o *WSAgent* e os demais trabalhos é que somente o *WSAgent* implementa persistência. Os demais *Frameworks* atuam como uma camada superior, deixando gerenciamento e persistência a cargo das aplicações que estão sendo integradas. Isso ocorre porque o ponto de integração do *WSAgent* é justamente as bases de dados, e não as aplicações. Isso também se mostra uma vantagem, uma vez que o código de aplicações legadas não necessita ser modificado.

Uma clara desvantagem do *WSAgent* frente às outras soluções apresentadas é o fato de não implementar mecanismos de segurança. O capítulo 6 traz essa questão como um ponto a ser trabalhado futuramente.

O comparativo entre as soluções de integração é resumido pela Tab. 2.2.

TABELA 2.2 – Comparativo entre o *WSAgent* e os trabalhos relacionados.

Característica	*WSAgent*	*HL7*	*CORBAmed*	*CCOW*
Aplica tecnologias de *Web Services*	✓	✓	×	×
Aplica Ontologias	✓	✓	×	×
É extensível	✓	✓	✓	✓
Implementa persistência	✓	×	×	×
Promove interoperabilidade	✓	✓	✓	×
Possui mecanismos de segurança	×	✓	✓	✓
Integração orientada a...	Serviços	Mensagens	Serviços	Usuário

Cabe ressaltar que a dimensionalidade do *WSAgent* frente às demais soluções é bastante diferente. A comparação apresentada, resumida na forma da Tab. 2.2, é feita com a intenção de facilitar a visualização das características que podem ser encontradas em cada solução, buscando contextualizar o *WSAgent*. No entanto, é sabido que o *Framework* não possui o mesmo nível de desenvolvimento e reconhecimento das demais, desenvolvidas por grupos de renome e já consolidadas no mercado; não há pretensão de elevar o *WSAgent* ao mesmo patamar.

O capítulo a seguir descreve a solução desenvolvida de forma mais detalhada.

4 *WSAGENT*

Este capítulo apresenta o *WSAgent*, que constitui uma proposta para solução de integração de bases heterogêneas fortemente baseada em tecnologias de *Web Services*. A solução envolve a criação de um *Framelet* (PREE, 1999b) de persistência e o desenvolvimento de Ontologias e Agentes de *Software* para manipulação dos dados trocados entre diferentes sistemas.

As principais características da solução e alguns modelos são apresentados a seguir.

4.1 ARQUITETURA

O *WSAgent* possui uma arquitetura única, mostrada na Fig. 4.1.

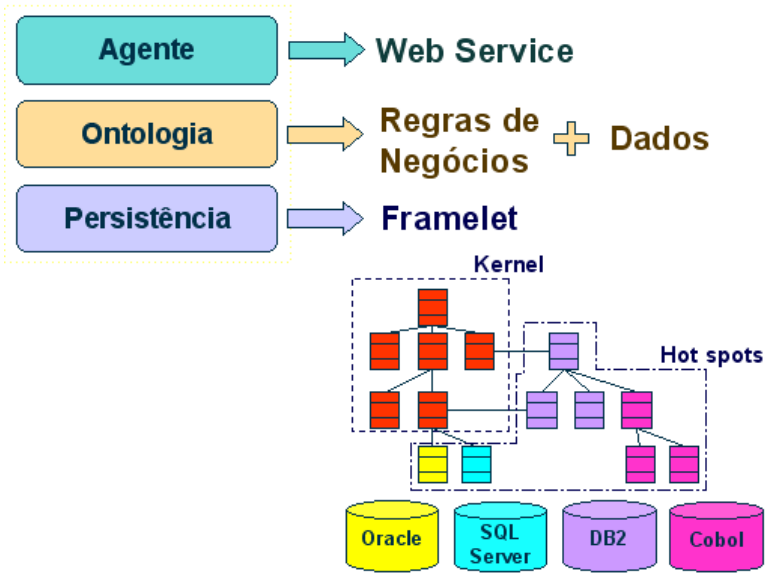


FIGURA 4.1 - Arquitetura do *WSAgent*.

Esta arquitetura é constituída, basicamente, de três camadas:

- camada de Agentes – utiliza *Web Services*, que devem atuar no papel de Agentes de *Software* para efetuar a comunicação entre os sistemas;
- camada de Ontologia – representa dados através de Ontologias e contém as regras de negócio que serão utilizadas para manipular dados trocados entre sistemas;
- camada de persistência – é constituída de um *Framelet* (um pequeno *Framework*), que será responsável pela parte de persistência dos dados trafegados entre os sistemas envolvidos, visando o armazenamento destas informações de forma correta em diversos bancos de dados.

A arquitetura é dita única porque o funcionamento dos Agentes de *Software* é guiado pelas informações definidas nas Ontologias. Assim, um Agente que está apto a trabalhar com o processo paciente-leito, por exemplo, pode ser configurado para trabalhar com o processo de estoque de medicamentos simplesmente ao ser "plugada" uma nova camada de Ontologia que define a estrutura e as regras para manipulação destes dados específicos. A camada de persistência a ser usada para todos os Agentes de *Software* é a mesma. Por tratar-se de um *Framelet*, é possível parametrizar o banco de dados com o qual se deseja trabalhar. E, se necessário, pode-se extender este *Framelet* para atender à necessidade de trabalho com algum banco de dados ainda não contemplado pela solução.

Há dois tipos de Agentes presentes na arquitetura: o primeiro age como mestre (ao ser invocado pela aplicação que realiza uma operação crítica, realiza algumas operações e então invoca os serviços de um segundo Agente, aguardando sua resposta); o segundo é um auxiliar do primeiro na realização das tarefas de atualização das bases de dados. Os dois tipos de Agentes compartilham da mesma arquitetura, cujo contexto de utilização é ilustrado pela Fig. 4.2.

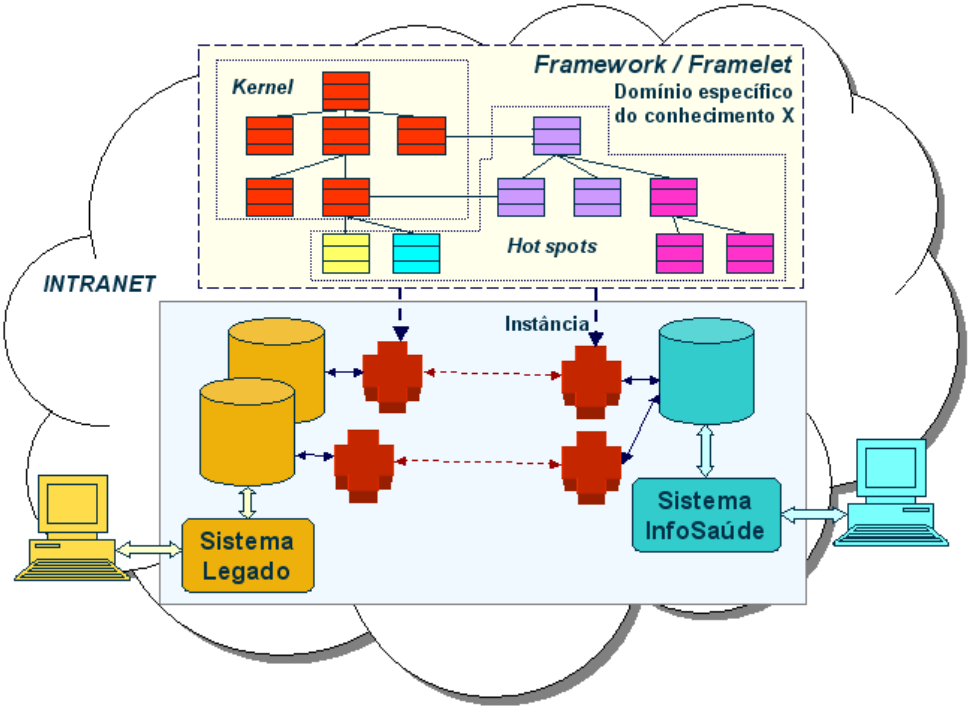


FIGURA 4.2 – Contexto de utilização da arquitetura proposta.

Assim, a solução global a ser utilizada por uma organização consiste de diversas instâncias desta arquitetura, ou seja: diversos Agentes de *Software* configurados com Ontologias específicas, parametrizados para manipulação de um ou mais tipos de bancos de dados e ligados a pontos específicos de aplicações que deverão ativar o funcionamento destes Agentes quando uma operação crítica for realizada[54].

Os Agentes não são inteligentes. A "agentificação" dos *Web Services* é caracterizada com base nas colocações de Wagner (G., 2003) a respeito de sistemas agentificados:

- perceber condições dinâmicas no ambiente – refere-se a mensagens que representam eventos de comunicação (para os *Web Services*, recebimento de uma requisição/resposta);
- agir para afetar as condições no ambiente – refere-se a ações decorrentes de comunicação (para os *Web Services*, atualização de tabelas de dados);
- raciocinar para interpretar percepções, resolver problemas, inferenciar e determinar ações – refere-se a processar corretamente as mensagens recebidas, responder

[54] Uma operação crítica é aquela que torna necessária a utilização de dados 100% atualizados.

corretamente a consultas e determinar ações apropriadas (para os *Web Services*, não seleção de dados que já foram devidamente atualizados);

- tratar itens de informação como crenças ou conhecimento (para os *Web Services*, aplicação do conceito de Ontologia);
- adicionar componentes de percepção (para os *Web Services*, mecanismos de invocação/resposta);
- prover suporte para comunicação entre ao Agentes, com base em uma linguagem padronizada (para os *Web Services*, uso do protocolo SOAP com mensagens baseadas em *XML*).

Os Agentes não precisam estar todos na mesma máquina ou servidor de aplicações, e nem associados a um mesmo banco de dados. Isto permite a total distribuição da solução, oferecendo grande flexibilidade na sua utilização.

Praticamente não há necessidade de alteração de código nos sistemas envolvidos. Como a solução opera diretamente sobre a camada de persistência, apenas é necessário customizá-la quanto ao tipo de banco de dados utilizado. Também é necessário realizar as chamadas aos Agentes na aplicação cuja base será atualizada (no contexto da empresa InfoSaúde, na aplicação Clínico Assistencial). A aplicação Legada permanece intacta.

Outra característica importante da arquitetura é que ela utiliza tecnologias *Java* em sua implementação – plataforma de desenvolvimento, linguagem de programação, uso de *drivers*, entre outros. Estas tecnologias oferecem diversos recursos para o trabalho com *XML*, *Web*, *Web Services*, redes, bancos de dados e independência do sistema operacional utilizado pelo usuário final. Ainda, oferecem independência de plataforma para que o usuário possa utilizar a solução sem necessitar um sistema operacional específico para utilizá-la. Assim, o custo da solução se torna reduzido.

A seguir, são descritos alguns modelos definidos a partir de informações obtidas junto à empresa InfoSaúde, e que descrevem como a solução proposta deve funcionar. Estes modelos são descritos a seguir usando a *UML* (BOOCH, 1999).

4.2 MODELO DE CASOS DE USO

Este modelo especifica as operações realizadas pelos Agentes de *Software* no contexto do problema InfoSaúde, e é mostrado na Fig. 4.4.

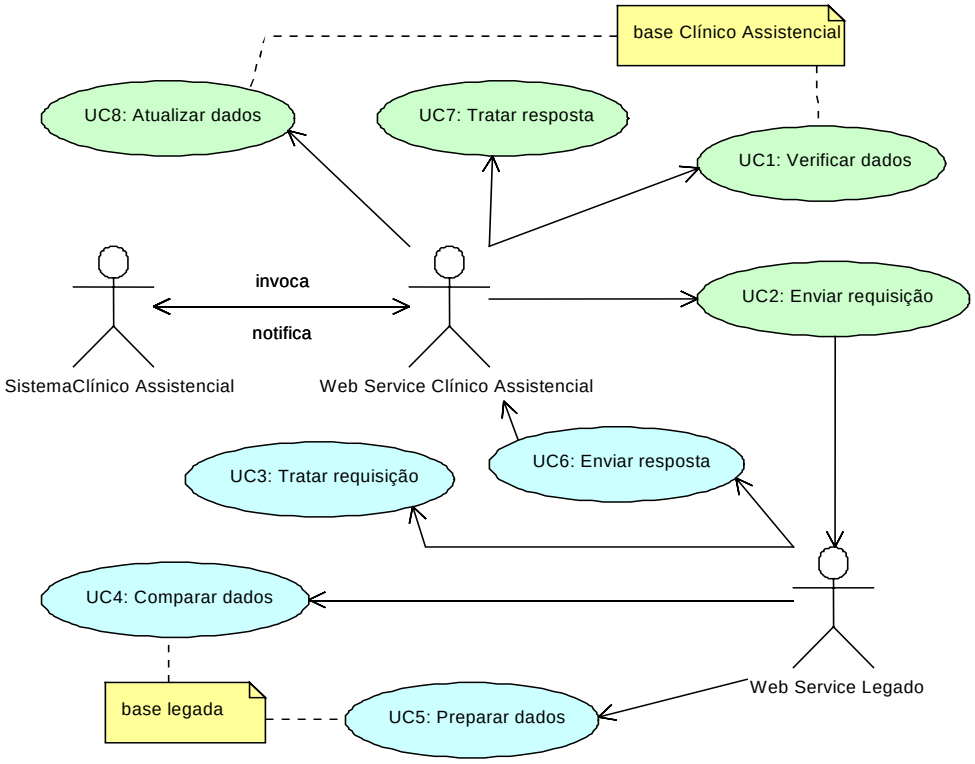


FIGURA 4.3 – Modelo de casos de uso.

Segue uma breve descrição de cada caso de uso do modelo:

- UC1: Verificar dados – o *Web Service* invocado pelo sistema Clínico Assistencial obtém as informações sobre atualização da base de dados, para verificar se é necessário invocar o outro Agente (uma operação crítica pode ser executada

imediatamente após outra, não havendo necessidade de nova atualização). “Onde buscar as informações” está definido na camada de Ontologia e regras de negócio;

- UC2: Enviar requisição – a mensagem de chamada é enviada para o *Web Service* vinculado ao sistema Legado;
- UC3: Tratar requisição – o *Web Service* vinculado ao sistema Legado recebe a chamada e inicia o processo de busca de dados;
- UC4: Comparar dados – o *Web Service* vinculado ao sistema Legado obtém os dados que foram recentemente modificados na base da aplicação Legada. De forma semelhante ao caso de uso UC1, “onde buscar as informações” está definido na camada de Ontologia e regras de negócio.
- UC5: Preparar dados – havendo dados mais atuais, o *Web Service* vinculado ao sistema Legado converte os dados de interesse em arquivo *XML*. Caso contrário, é enviado aviso de que não há dados atualizados;
- UC6: Enviar resposta – o *Web Service* vinculado ao sistema Legado envia o arquivo *XML* gerado para o *Web Service* vinculado ao sistema Clínico Assistencial;
- UC7: Tratar resposta – o *Web Service* vinculado ao sistema Clínico Assistencial recebe a resposta (o arquivo *XML* com os dados ou o aviso de que não há atualização a ser feita) e preocessa seu conteúdo;
- UC8: Atualizar dados – caso haja alterações a fazer, o *Web Service* vinculado ao sistema Clínico Assistencial realiza a atualização da sua base de dados a partir do arquivo *XML* recebido, também em conformidade com a camada de Ontologia e regras de negócios, para compatibilizar todas as informações recebidas e tratá-las da forma correta.

Ao final do processo, o controle é retornado à aplicação Clínico Assistencial.

Em todas as etapas que envolvem pesquisa de dados, bem como operações de alteração, inserção e deleção, atua a camada de persistência. Ao preparar o Agente para ser vinculado a uma operação específica do sistema InfoSaúde, o *Framelet* deve ser parametrizado para uso do banco de dados do sistema (deve-se "gerar" a solução para determinado banco da dados). Assim, quando as operações descritas anteriormente forem executadas, o Agente vinculado ao sistema Clínico Assistencial terá a camada de persistência trabalhando em conformidade com o sistema InfoSaúde. O mesmo vale para o Agente vinculado ao sistema Legado.

4.3 MODELO DE INTERAÇÕES

O modelo apresentado na Fig. 4.3 mostra o funcionamento da solução de forma seqüencial, inserido no contexto do problema da InfoSaúde.

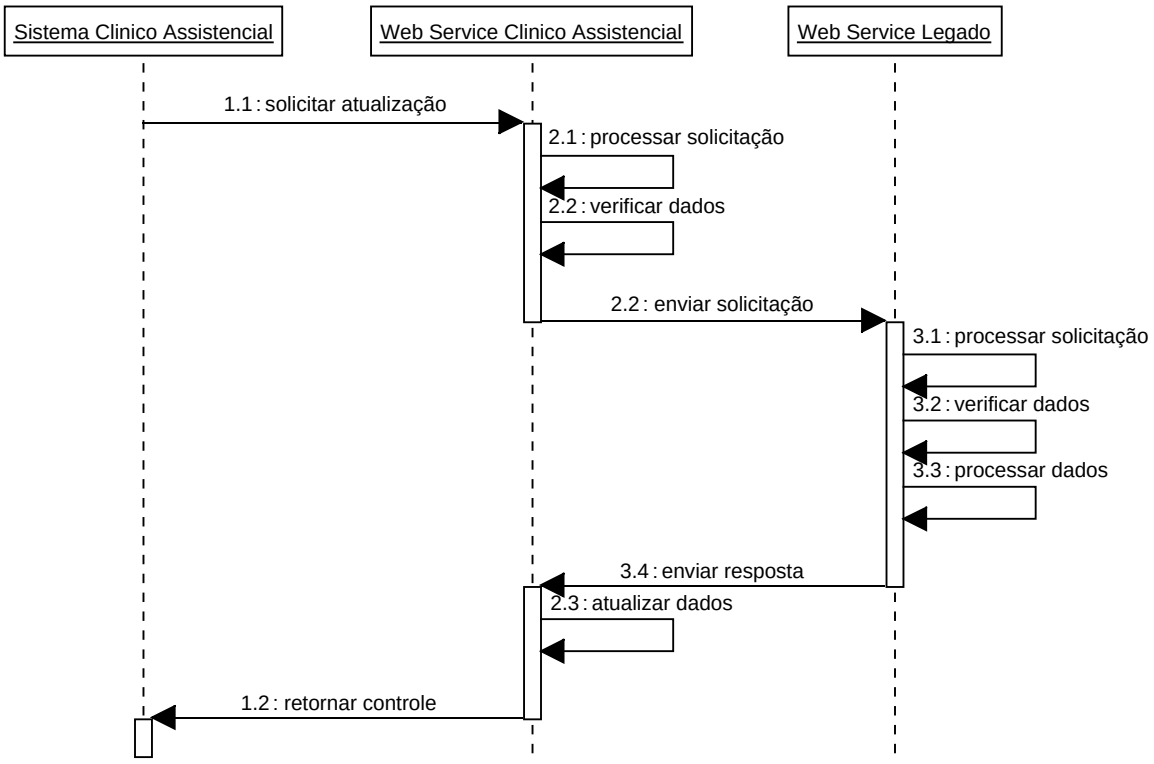


FIGURA 4.4 – Modelo de interações: diagrama de seqüência.

Os *Web Services* envolvidos na solução são síncronos (SUN MICROSYSTEMS, 2002). A partir da solicitação de um *Web Service* associado ao sistema Clínico Assistencial a outro *Web Service* associado ao banco de dados Legado, o primeiro permanece esperando uma resposta antes de continuar seu processamento.

Assim, ao ser executada uma operação crítica no sistema Clínico Assistencial, o *Web Service* correspondente é invocado pelo sistema e verifica os dados presentes na base da aplicação. A partir disto, invoca o *Web Service* vinculado ao sistema Legado, que analisa a requisição, verifica a existência de novos dados, prepara estes dados e os envia ao *Web Service* solicitante. No caso de não haver dados a atualizar, uma resposta adequada é enviada. A partir disto, o *Web Service* vinculado ao sistema Clínico Assistencial analisa a resposta recebida e, se for o caso, atualiza as informações notificando a aplicação de que o processo foi concluído.

Esta forma de realizar a pesquisa por dados atualizados – a partir de operações críticas no sistema – evita uma sobrecarga de tráfego na rede. Como pode haver fluxos mais pesados de informações sendo inseridas via sistema Legado, um simples monitoramento da base legada causaria problemas de desempenho e funcionamento do sistema como um todo.

A metodologia para definição dos aspectos críticos é algo particular à empresa que fará uso da solução, ou seja, depende da aplicação que irá fazer uso dos *Web Services* e das informações que a empresa considera como imprescindíveis no momento da atualização. Se duas empresas fizerem uso da solução, ou se a mesma empresa fizer uso da solução em clientes diferenciados, as operações consideradas críticas podem apresentar diferenças.

4.4 MODELO DE CLASSES

A Fig. 4.5 e a Fig. 4.6 mostram o modelo de classes do *WSAgent*.

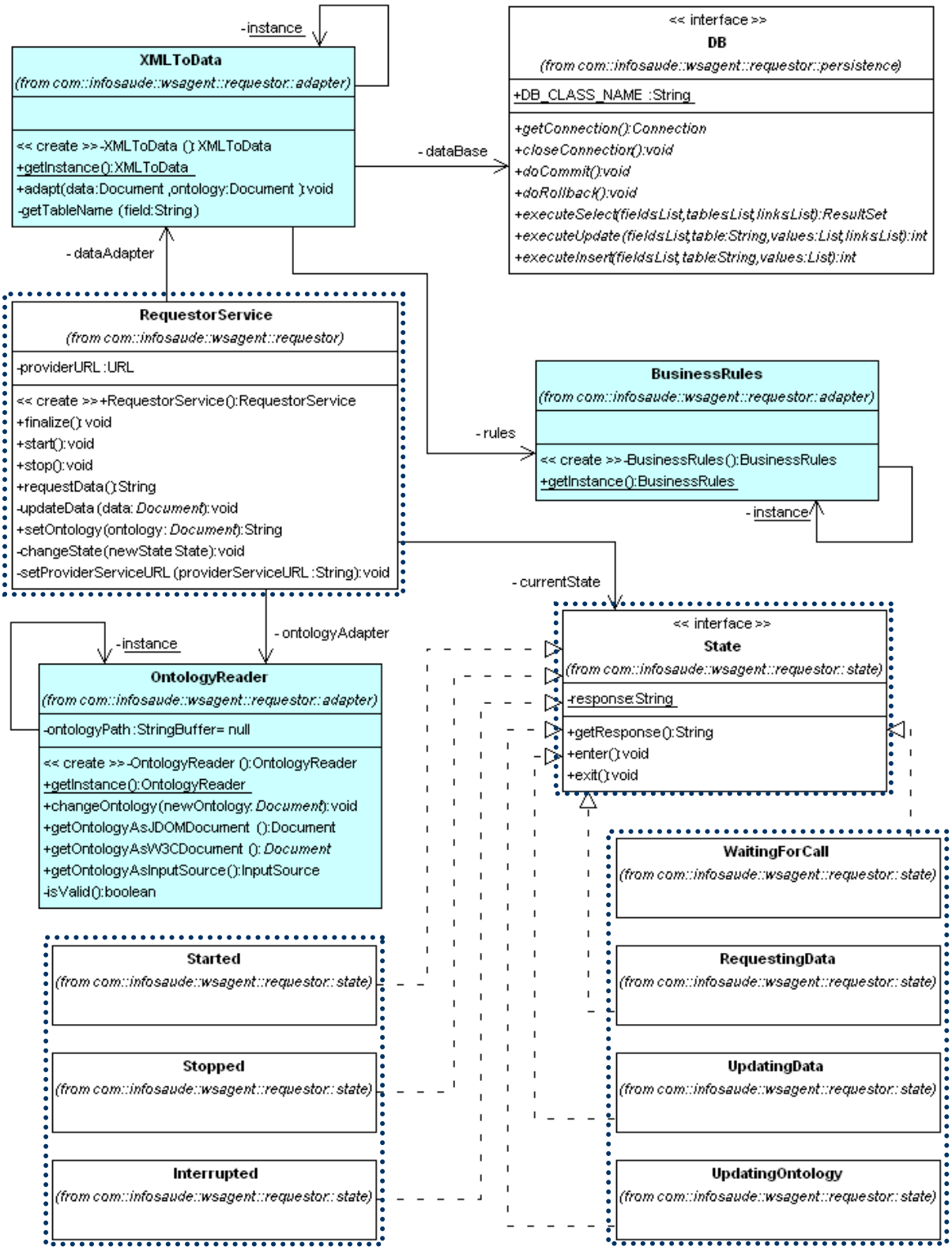


FIGURA 4.5 – Modelo de classes para o Agente *Requestor*.

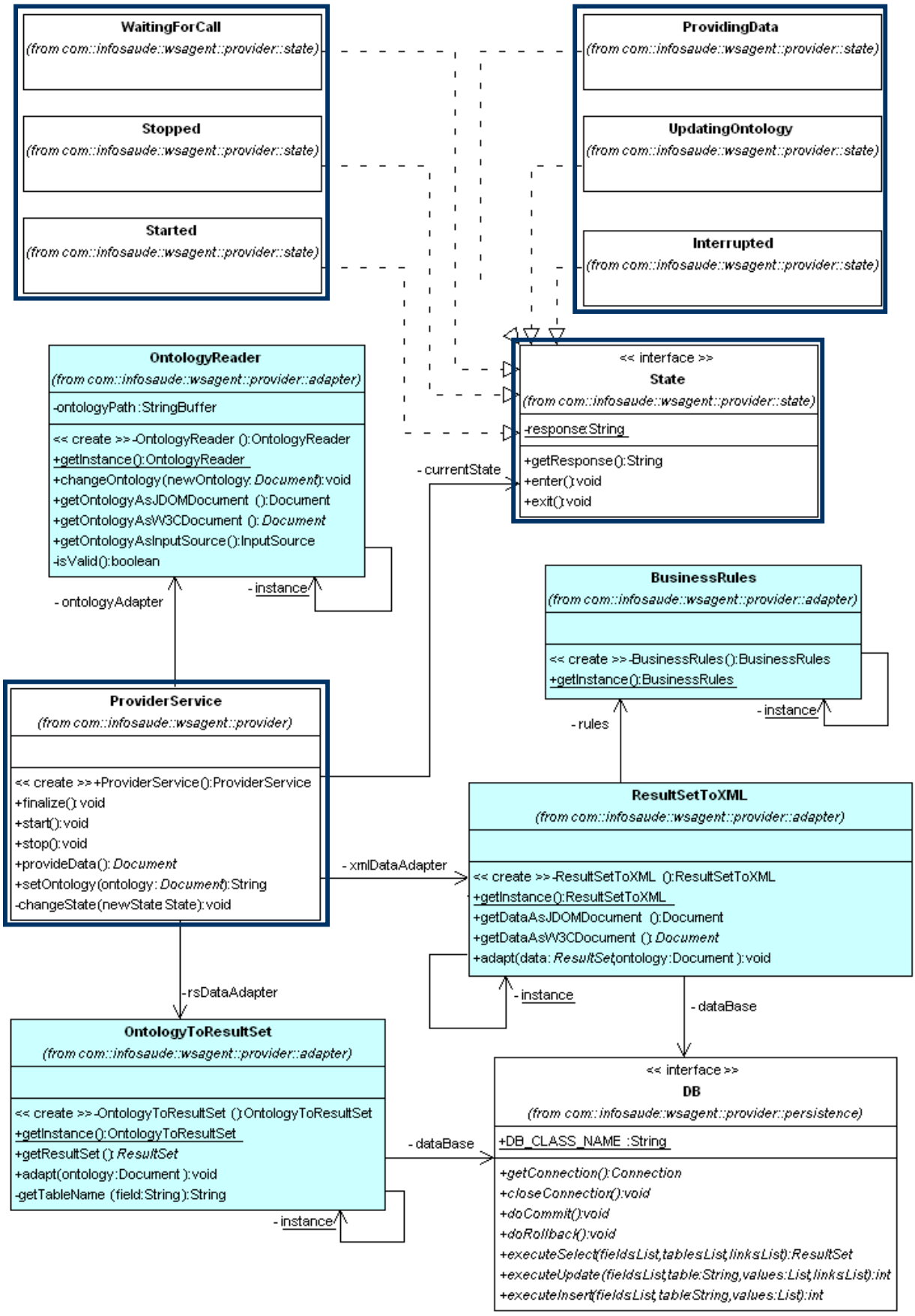


FIGURA 4.6 – Modelo de classes para o Agente *Provider*.

A solução consiste de dois tipos de Agentes: o primeiro, que age como mestre, foi chamado Requestor, e as classes que o compõem são mostradas na Fig. 4.5; o segundo, que age sob solicitação do primeiro, foi chamado *Provider*, e as classes que o compõem são mostradas na Fig. 4.6.

A classe `RequestorService` corresponde ao Agente (*Web Service*) que será invocado pela aplicação para dar início ao processo de atualização. Esta classe possui métodos para definir a Ontologia que será utilizada, dar seqüência ao processo de atualização dos dados (contatando o outro *Web Service* para isso) e efetuar a atualização dos dados (armazenando o último momento em que os dados foram atualizados, para controlar o intervalo entre as atualizações).

A classe `ProviderService` corresponde ao Agente (*Web Service*) que está ligado à base legada. Esta classe possui métodos para definir a Ontologia que será utilizada, realizar a verificação dos dados que foram inseridos ou modificados e dar seguimento ao processo de atualização dos dados, enviando-os ao *Web Service* que fez a solicitação.

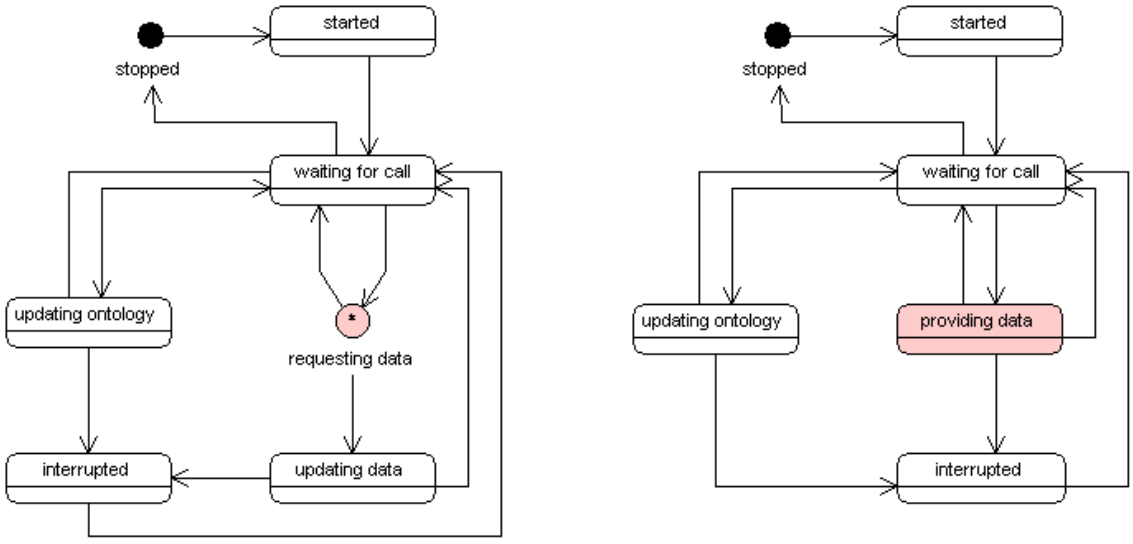


FIGURA 4.7 – Diagramas de estados para os Agentes *Requestor* e *Provider*.

A Fig. 4.7 mostra os diagramas de estados referentes ao funcionamento dos Agentes. O modelo é complementado pela interface `State` e as classes que implementam esta interface:

`Started`, `Stopped`, `WaitingForCall`, `RequestingData`, `ProvidingData`, `UpdatingData`, `UpdatingOntology` e `Interrupted`. Este conjunto de classes, que na Fig 4.5 e na Fig. 4.6 aparecem envolvidas por uma linha pontilhada, correspondem ao *Design Pattern State* (GAMMA, 1995) e representam os estados pelos quais cada Agente pode passar durante seu funcionamento.

Na Fig. 4.7, o diagrama da esquerda corresponde ao Agente *Requestor*. Ele pode, quando ”parado”, ser iniciado e então permanecer aguardando por chamadas. Estas chamadas podem ser para que o Agente atualize a Ontologia que utiliza, quando esta precisar ser modificada, para que o Agente inicie o processo de atualização de dados, ou mesmo para tirá-lo de operação. O estado `requesting data` é síncrono, pois envolve chamada ao outro Agente, aguardando sua resposta para dar seguimento às operações. Quando requisitando dados, após receber a resposta do Agente *Provider*, o Agente *Requestor* passa a atualizar esses dados. Após ter atualizado os dados ou a ontologia, o Agente passa novamente a aguardar chamadas. Enquanto está realizando alguma operação, podem ocorrer erros; neste caso, o Agente passa ao estado `interrupted`, onde o erro é tratado e, em seguida, volta a esperar por chamadas.

Na Fig. 4.7, o diagrama da direita corresponde ao Agente *Provider*. Da mesma forma que o Agente *Requestor*, ele pode, quando ”parado”, ser iniciado e então permanecer aguardando por chamadas. Estas chamadas podem ser para que o Agente atualize a Ontologia que utiliza, quando esta precisar ser modificada, para que o Agente busque dados que foram modificados ou até mesmo para tirá-lo de operação. O estado `providing data` são chamados pelo outro Agente, que aguarda sua resposta para dar seguimento às operações. Ao enviar resposta para o Agente *Requestor*, o Agente *Provider* passa novamente a aguardar chamadas. Enquanto está realizando alguma operação, podem ocorrer erros; neste caso, o

Agente passa ao estado `interrupted`, onde o erro é tratado e, em seguida, volta a esperar por chamadas.

Na Fig. 4.5 e na Fig 4.6, algumas classes são destacadas com cor de preenchimento. São elas:

- `OntologyReader` – faz a leitura e validação do arquivo de Ontologia, disponilizando métodos para obter esse arquivo em alguns formatos diferentes;
- `BusinessRules` – contém métodos referentes às regras de negócios, ou seja, operações que devem ser executadas com os dados obtidos nas bases de dados, a fim de obter a informação no formato necessário para atualização (por exemplo, um valor obtido em uma base de dados, sobre o qual vários cálculos devem ser efetuados antes que o valor resultante desses cálculos seja utilizado para atualização);
- `OntologyToResultSet` – faz uso do arquivo de Ontologia convertido para obter o conjunto de registros que devem ser selecionados na base de dados;
- `ResultSetToXML` – faz com que os dados obtidos na base legada sejam representados em formato *XML*, para que sejam enviados ao *Web Service* que fez a requisição;
- `XMLToData` – adapta os dados recebidos para o formato de consultas *SQL* e realiza a atualização da base de dados.

Estas classes foram idealizadas segundo o *Design Pattern Singleton* (GAMMA, 1995), que garante haver apenas uma instância de cada classe durante a execução. Assim, é possível controlar de forma mais adequada a manipulação da Ontologia e de arquivos de dados que são trocados pelos Agentes.

Estas mesmas classes, com exceção de `BusinessRules`, também correspondem ao *Design Pattern Adapter* (GAMMA, 1995) que, como o nome diz, permite a adaptação das informações de um formato para outro. Por exemplo, `ResultSetToXML` adapta os dados que estão em um formato específico da linguagem Java (um *ResultSet* corresponde a um conjunto

de dados obtidos em uma base de dados) para o formato XML, de forma que possam ser trocados entre os Agentes.

As classes `BusinessRules` e `OntologyReader`, em conjunto com os arquivos *XML* que representam a Ontologia, constituem a camada de Ontologias e regras de negócio da solução.

A camada de persistência é formada pela interface `DB` e pelas classes `OntologyToResultSet`, `XMLToData` e `ResultSetToXML`. A interface `DB` deve ser implementada para criação de classes específicas para cada tipo de banco de dados a ser utilizado. Cada nova classe torna-se responsável por realizar a conexão com a base de dados e executar *scripts* de consulta e atualização. Esta classe não é utilizada diretamente pelos *Web Services*, mas sim pelas classes adaptadoras.

As classes que compõem o *Framework* foram organizadas segundo a seguinte estrutura de pacotes:

- `com.infosaude.wsagent.provider` – classe principal do Agente *Provider*;
- `com.infosaude.wsagent.provider.state` – classes de estados do Agente;
- `com.infosaude.wsagent.provider.adapter` – classes adaptadoras e de regras de negócio;
- `com.infosaude.wsagent.provider.persistence` – classes de persistência;
- `com.infosaude.wsagent.requestor` – classe principal do Agente *Requestor*;
- `com.infosaude.wsagent.requestor.state` – classes de estados do Agente;
- `com.infosaude.wsagent.requestor.adapter` – classes adaptadoras e de regras de negócio;
- `com.infosaude.wsagent.requestor.persistence` – classes de persistência.

A Fig. 4.8 destaca as classes que formam o *kernel* da solução (com cor de preenchimento) e aquelas que oferecem pontos de flexibilização (sem cor de preenchimento).

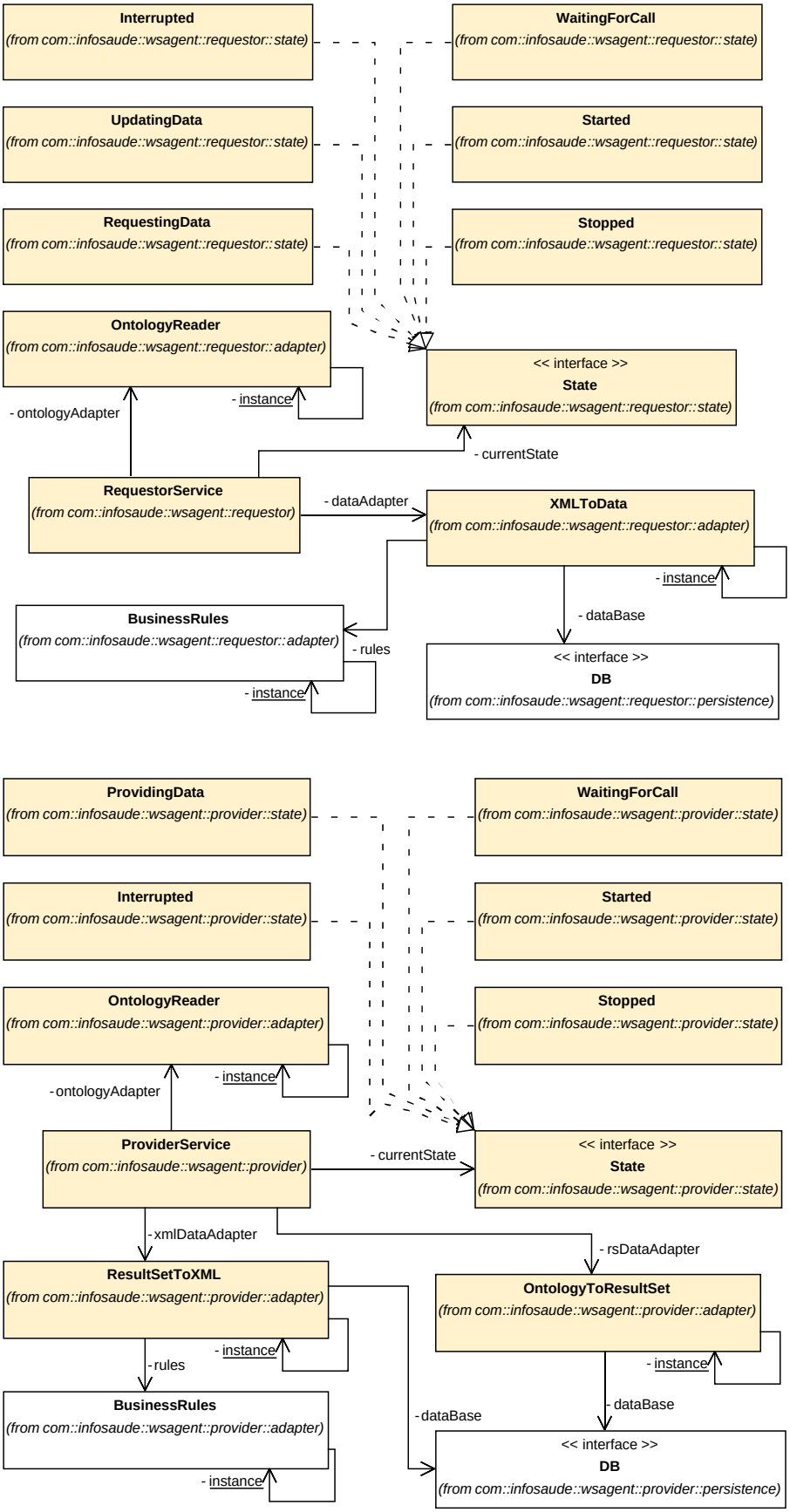


FIGURA 4.8 – *Kernel* e pontos de flexibilização (*hot spots*).

É possível flexibilizar a solução através da adição de métodos na classe `BusinessRules`, os quais são referenciados no arquivo de Ontologia e executados durante o processo de atualização dos dados. Outra forma de flexibilização é a implementação da interface `DB` para criar classes próprias para cada tipo de banco de dados a ser utilizado. E também é possível flexibilizar a solução através da modificação do arquivo de Ontologia, que deve ser repassado aos Agentes para que façam a atualização.

4.5 ARQUIVOS DE ONTOLOGIA

As informações obtidas em diferentes bases de dados são compatibilizadas através do uso de Ontologias. Por exemplo, na base de dados do sistema Clínico Assistencial, o nome do paciente pode ser encontrado no campo `NOME` da tabela `CA_PACIENTE`, e possui tipo `VARCHAR2` de tamanho 60. Em um sistema Legado, a mesma informação pode estar contida em uma tabela de nome diferente, separada em campos que contêm o primeiro nome, o nome do meio e o sobrenome, separadamente. O sexo do paciente pode estar representado como um único caracter (`M` ou `F`) em uma base de dados e, na outra, estar representado de forma abreviada (`MAS`, `FEM`). Os formatos de datas, em se tratando da data de nascimento do paciente, podem estar definidas em formatos diferenciados.

A Ontologia fornece uma estrutura padrão que deve ser extendida para contemplar as informações sobre a localização de cada uma das propriedades da classe *Paciente*. Assim, o Agente vinculado a cada base de dados deve utilizar uma Ontologia customizada, contendo os campos onde as informações podem ser encontradas. Cabe ressaltar que o conceito de Ontologia foi adaptado para uso no *WSAgent*. Ele foi utilizado para criar os conceitos que indicarão aos Agentes onde encontrar as informações, porém isto foi feito na forma de um mapeamento da base de dados, sem conter as referências que os formatos de representação de

Ontologias utilizam. Este formato simplificado foi ao encontro das necessidades de desenvolvimento da solução para o momento.

Para auxiliar na definição dos arquivos de Ontologia, foi desenvolvido um gerador (apresentado na Fig. 4.9, na Fig. 4.10 e na Fig. 4.11).

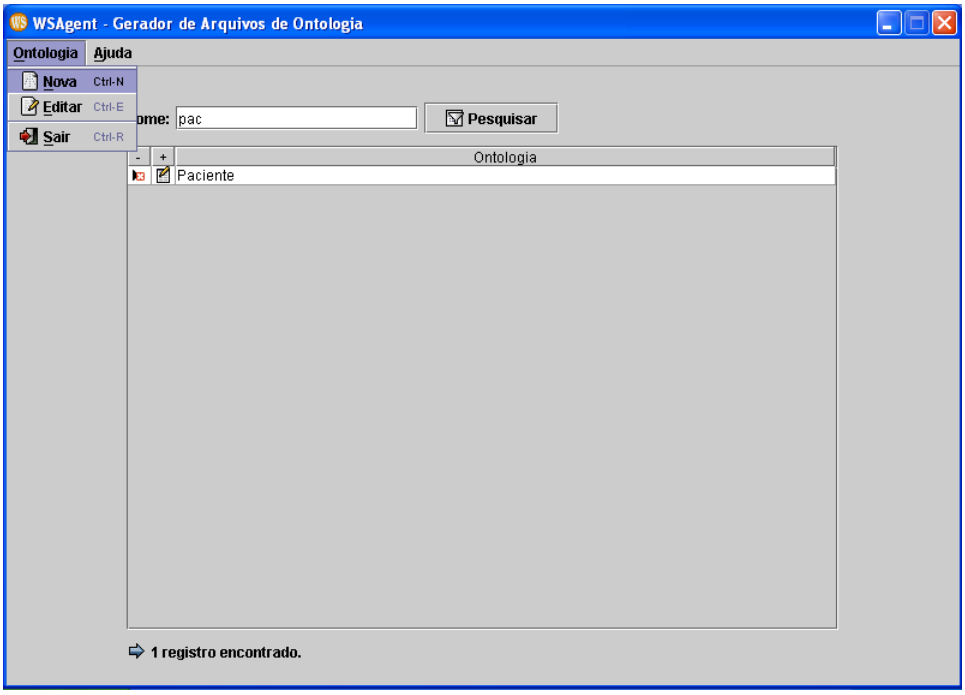


FIGURA 4.9 – Gerador de arquivos de Ontologia (modo de consulta).

O gerador armazena a definição das Ontologias em uma base de dados, para que as mesmas sejam recuperadas posteriormente. A Fig. 4.9 mostra a tela de consulta das Ontologias cadastradas. Nesta tela, podemos excluir uma Ontologia ou ativar o modo de edição, trazendo os dados referentes à mesma.

A partir das informações fornecidas na tela da Fig. 4.10, é possível gerar os arquivos *XML* para os dois Agentes. Esses arquivos, posteriormente, são enviados para os Agentes via *SOAP*, ficando armazenados internamente em suas classes.

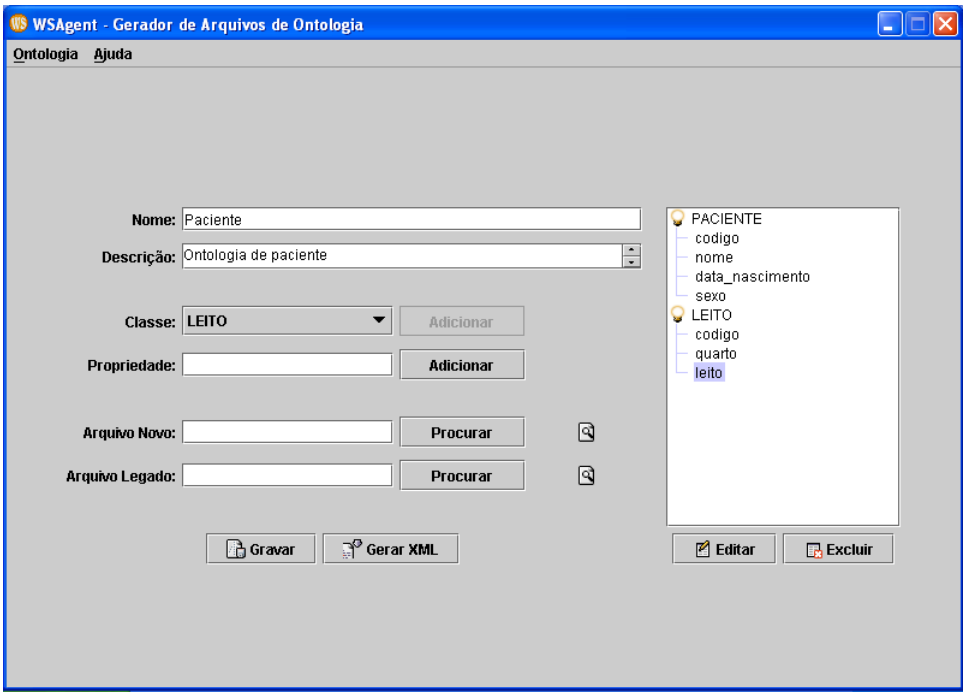


FIGURA 4.10 – Gerador de arquivos de Ontologia (modo de criação/edição).

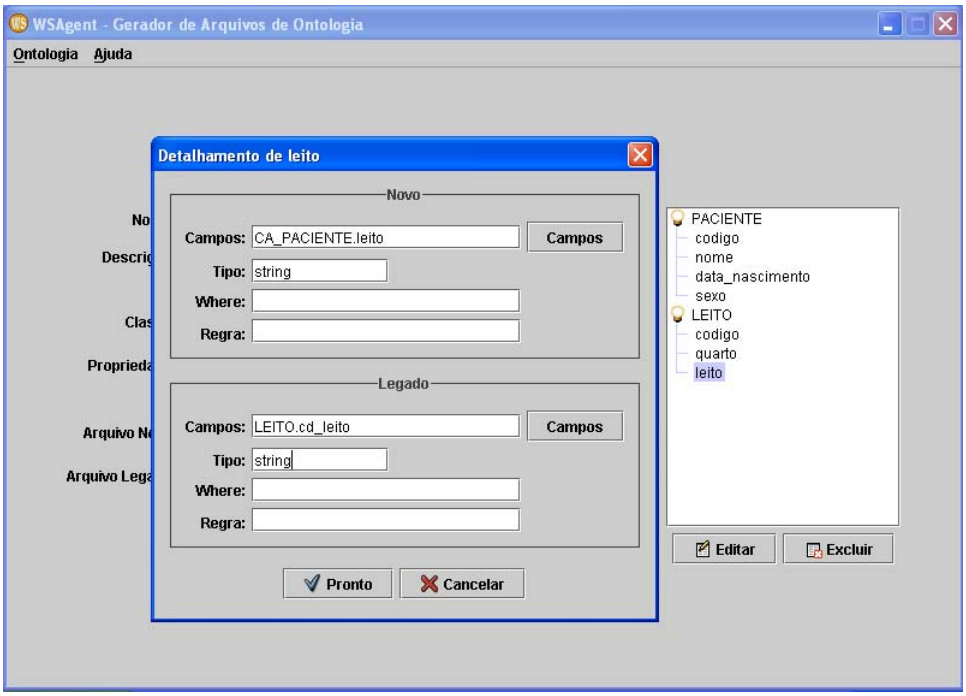


FIGURA 4.11 – Gerador de arquivos de Ontologia (detalhamento de propriedade).

As classes definem as entidades sobre as quais se deseja obter as informações. Cada classe possui propriedades, que são as informações que se deseja obter. Em cada propriedade

são definidos: as tabelas e campos de onde as informações serão obtidas, o tipo genérico dessa informação, e, opcionalmente, a condição de pesquisa na base de dados e a regra para manipulação da informação. Esse detalhamento é feito na tela da Fig. 4.11.

Essas informações, que devem gerar os arquivos de ontologia, são validadas através de *XML Schema*. Além de definir a estrutura que um arquivo de Ontologia deve apresentar, com o tipo de dados e o número possível de ocorrências de cada elemento, o *schema* coloca a obrigatoriedade de um arquivo *XML* de Ontologia conter pelo menos um de cada elemento: ontologia, classe e propriedade. O conteúdo referente ao *XML Schema* é mostrado a seguir.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="ontology">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="class" minOccurs="1" maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="property" minOccurs="1"
                                              maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="field" minOccurs="1"
                                             maxOccurs="unbounded"/>
              <xs:element name="type" minOccurs="1" maxOccurs="1"/>
              <xs:element name="where" minOccurs="0" maxOccurs="1"/>
              <xs:element name="rule" minOccurs="0" maxOccurs="1"/>
            </xs:sequence>
          <xs:attribute name="name" type="xs:string" use="required"/>
          </xs:complexType>
          </xs:element>
        </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
      </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  </xs:element>
</xs:schema>
```

Exemplos de arquivos de Ontologia construídos com base neste *schema* são mostrados no capítulo a seguir, que trará um estudo de caso sobre a construção de um protótipo do *WSAgent*.

5 ESTUDO DE CASO

Este capítulo trata do desenvolvimento de um protótipo do *WSAgent*.

A empresa InfoSaúde pretende utilizar o *WSAgent* para complementar o sistema que hoje vem sendo desenvolvido para o Hospital Moinhos de Vento (localizado em Porto Alegre) e que será comercializado para outros hospitais, após concluído. Esse sistema chama-se Clínico Assistencial, e foi projetado para trabalhar com mais de um sistema operacional, banco de dados e idioma.

O *WSAgent* constitui um diferencial para o sistema Clínico Assistencial, pois permite extender suas características multi-idioma, multi-banco, etc. Além disso, permitirá que qualquer outro sistema seja integrado ao sistema Clínico Assistencial. Devido a esse fator, a empresa deseja investir na continuidade do trabalho. Para que a solução possa ser comercializada, primeiro o *WSAgent* precisa tornar-se um produto completo.

O sistema Clínico Assistencial encontra-se ainda em desenvolvimento. Uma versão dele, ainda não completa, está em teste junto à equipe do Hospital Moinhos de Vento. O sistema está sendo construído com base em tecnologias *Java 2 Enterprise Edition* (*J2EE*), utilizando servidor de aplicações *JBoss*[55] e base de dados *Oracle*[56].

Buscou-se guiar o desenvolvimento do *WSAgent* de forma compatível com o sistema Clínico Assistencial, fazendo uso de tecnologias e ferramentas baseadas em Java e de uso gratuito.

[55] Informações podem ser obtidas em: <http://www.jboss.org>
[56] Informações podem ser obtidas em: <http://www.oracle.com>

5.1 TECNOLOGIAS UTILIZADAS

Para o desenvolvimento do protótipo do *WSAgent*, foram utilizadas tecnologias *Java*. Além da própria linguagem, foram utilizadas ferramentas disponíveis na *Internet*, de uso gratuito, compatíveis com *Java*.

O ambiente de desenvolvimento escolhido foi o *Eclipse*[57]. Esta ferramenta permite trabalhar com *Java* para desenvolvimento de programas simples, bem como de nível comercial. É possível integrar diversos *plugins* e *APIs* para facilitar o trabalho, por exemplo, com *J2EE*. Esta ferramenta também está disponível no HMV, junto à equipe da InfoSaúde.

Para desenvolvimento da camada de *Web Services*, foi necessário trabalhar com *SOAP*, o protocolo de comunicação utilizado por este grupo de tecnologias. Em lugar de desenvolver as funcionalidades a partir do zero, utilizando recursos do pacote *JWSDP*, optou-se por utilizar alguma ferramenta disponível gratuitamente. A escolha foi o *Axis*[58], um *Framework* desenvolvido pela *Apache* para construir *software* que possa processar a troca de mensagens *SOAP* (clientes, servidores, *gateways*, etc). Além de *APIs* para o desenvolvimento de *software*, o *Axis* provê o mecanismo para a disponibilização dos *Web Services* em um servidor de aplicações. Para o desenvolvimento do protótipo, foi utilizada a versão 1.1 da ferramenta.

O servidor de aplicações escolhido foi o *JBoss*, o mesmo utilizado para disponibilizar o sistema Clínico Assistencial da InfoSaúde. Testes foram realizados com versões 3.2.x e 4.0.x.

Também foram necessários *parsers* para manipular arquivos *XML*. Optou-se por trabalhar com SAX, mais indicado na literatura por questões de desempenho, através do uso da API *Java Document Object Model* (*JDOM*[59]). *JDOM* facilita o trabalho de criação/manipulação de estruturas XML.

---

[57] Informações podem ser obtidas em: <http://www.eclipse.org>
[58] Informações podem ser obtidas em: <http://ws.apache.org/axis/>
[59] Informações podem ser encontradas em: <http://www.jdom.org>

O acesso a bases de dados foi realizado com *drivers Java Database Connectivity* (*JDBC*), que podem ser adquiridos até de forma gratuita, via *download* em *sites* de empresas.

5.2 CAMADA DE AGENTES (*WEB SERVICES*)

As classes `RequestorService` e `ProviderService` implementam os mecanismos básicos de operação dos Agentes. Elas foram implementadas com base em *APIs* do *Apache Axis*, para que possam operar como *Web Services*, e realizam as chamadas a todas as classes adaptadoras. Estas, por sua vez, fazem chamadas às classes responsáveis pela conexão e execução de *scripts* nas bases de dados.

O mecanismo principal do Agente *Requestor* é a chamada ao Agente *Provider* e atualização dos dados segundo o arquivo *XML* recebido. Esta chamada é realizada de forma semelhante ao trecho[60] a seguir.

```
Service providerService = new Service();
Call providerCall = (Call)providerService.createCall();
providerCall.setTargetEndpointAddress(providerURL);
Document providerResponse = (org.w3c.dom.Document)
                            providerCall.invoke("provideData", null);
```

O Agente *Requestor* cria um serviço (`providerService`) e prepara a chamada ao mesmo (`providerCall`), fornecendo o endereço através do qual o Agente Provider pode ser localizado (`providerURL`). Ele então invoca o serviço, informando a operação que deseja que o Agente *Provider* execute (`provideData`) e o valor dos parâmetros que devem ser passados a essa operação (no caso, a operação não possui parâmetros).

O Agente *Provider* então realiza a busca de dados de forma semelhante ao trecho a seguir.

[60] Por questão de legibilidade, nos trechos de código apresentados neste capítulo, foram omitidas operações de tratamento de erros

```
ontologyAdapter = OntologyReader.getInstance();
rsDataAdapter = OntologyToResultSet.getInstance();
xmlDataAdapter = ResultSetToXML.getInstance() ;

(…)

rsDataAdapter.adapt(ontologyAdapter.getOntologyAsJDOMDocument());
xmlDataAdapter.adapt(rsDataAdapter.getResultSet(),
                        ontologyAdapter.getOntologyAsJDOMDocument());
Document data = xmlDataAdapter.getDataAsW3CDocument();
```

Aqui são invocadas as classes adaptadoras do Agente *Provider* (`OntologyReader`, `OntologyToResultSet` e `ResultSetToXML`). Sua operação conjunta produz um documento de dados em formato XML (`data`), que é enviado ao Agente *Requestor*.

Com base neste documento, o Agente *Requestor* inicia a atualização dos dados, de forma semelhante ao trecho a seguir.

```
ontologyAdapter = OntologyReader.getInstance();
dataAdapter = XMLToData.getInstance();

(…)

SAXBuilder builder = new SAXBuilder();
Org.jdom.Document dataDocument = builder.build(
                   New StringReader(XMLUtils.DocumentToString(data)));
dataAdapter.adapt(dataDocument,
                        ontologyAdapter.getOntologyAsJDOMDocument());
```

O documento recebido do Agente *Provider* (`data`) é convertido para o formato *JDOM* que a aplicação necessita, com base em *SAX*. O documento convertido (`dataDocument`) é então passado para a classe adaptadora (`XMLToData`) que irá processá-lo e atualizar a base de dados.

O mecanismo de atualização da Ontologia funciona de um Agente é bastante simples: o Agente é invocado, recebendo a nova Ontologia como parâmetro (`ontology`). Ele atualiza o conteúdo do seu arquivo de Ontologia com base no arquivo recebido, através da classe adaptadora (Ontology`Reader`, cuja instância é guardada em `ontologyAdapter`), como pode ser visto na linha a seguir.

```
ontologyAdapter.changeOntology(ontology);
```

A classe adaptadora então substitui o conteúdo do arquivo gravado em disco pelo conteúdo recebido via parâmetro.

Esses são alguns aspectos relativos à implementação dos Agentes. A sua instalação no servidor *JBoss*, para que trabalhe com *Axis*, exige um certo trabalho de configuração. É preciso instalar primeiro o *JBoss*, depois o *Axis*. Detalhes sobre como realizar cada instalação podem ser obtidos junto aos *sites* que disponibilizam as ferramentas para *download*.

Para disponibilizar um *Web Service*, deve-se criar um arquivo com extensão `jar` que contenha seus arquivos compilados, e então copiar este arquivo para o diretório `deploy` do tipo de servidor que está sendo utilizado. O JBoss disponibiliza três tipos de servidor: `minimal` (com um mínimo de recursos), `default` (é a opção padrão, mais utilizada) e `all` (todos os recursos disponíveis). Há outras opções de instalação que podem ser encontradas na documentação das ferramentas, mas esta foi considerada mais adequada para o trabalho com os Agentes neste momento.

As classes dos dois tipos de Agentes podem ser disponibilizadas no mesmo arquivo de extensão `jar` ou em arquivos separados. Podem permanecer no mesmo servidor de aplicações ou em servidores distintos, em diferentes máquinas.

Após disponibilizar os arquivos, é necessário incluir algumas linhas no arquivo de configuração `server-config.wsdd`, do *Axis*. Segue um trecho de exemplo:

```
<service name="RequestorService" style="wrapped">
  <parameter name="allowedMethods" value="requestData setOntology"/>
  <parameter name="enableRemoteAdmin" value="false"/>
  <parameter name="className"
           value="com.infosaude.wsagent.requestor.RequestorService"/>
  <parameter name="scope" value="application"/>
</service>
```

Basicamente, o que esse trecho diz é que temos um serviço chamado `RequestorService`, que disponibiliza as operações `requestData` e `setOntology`; a classe referente ao serviço

é `com.infosaude.wsagent.provider.ProviderService` e o estilo de comunicação utilizado é `wrapped`.

No *Axis*, há quatro estilos de comunicação que podem ser utilizados pelos *Web Services*:

- `RPC` – trata automaticamente o conteúdo das mensagens, realizando a conversão para tipos *Java* segundo convenções SOAP RPC, utilizando serialização;
- `document` – não aplica codificação como o tipo `RPC`, mas realiza alguma conversão de tipos entre *XML* e *Java*, tratando o corpo *SOAP* das mensagens como uma grande estrutura;
- `wrapped` – é como o tipo `document`, mas quebra o corpo *SOAP* das mensagens em parâmetros individuais;
- `message` – permite que o conteúdo *XML* das mensagens seja recebido de forma intocada, sem nenhuma codificação/conversão, de forma a ser tratado pelo *software* desenvolvido e não pelo mecanismo *SOAP* do *Axis*.

Na fase atual, o tipo `wrapped` mostrou-se adequado para troca tanto de documentos *XML* quanto de dados de tipos Java como `String`, entre outros. A documentação do *Axis* traz outras opções que podem ser utilizadas para criar esta configuração de serviço, que pode tornar-se mais complexa.

A Fig. 5.1 mostra o serviço já disponível no servidor *JBoss*, listando os métodos que podem ser acessados.

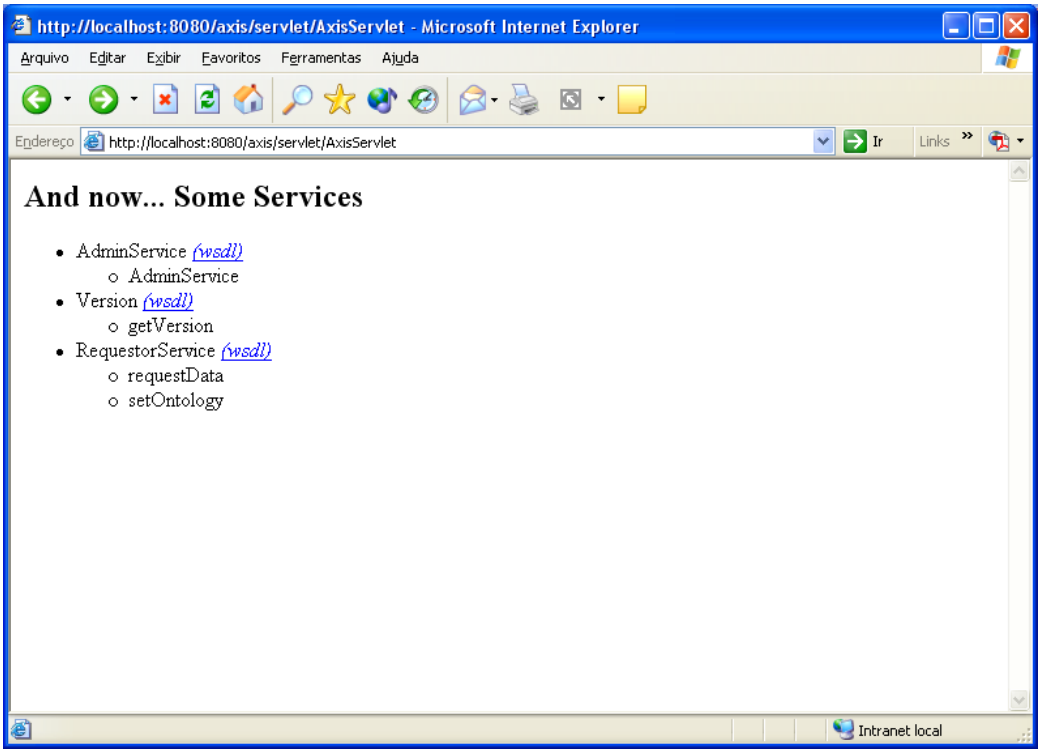


FIGURA 5.1 – Serviço disponibilizado no *JBoss* com *Axis*.

Com o *Axis* é possível obter automaticamante o arquivo *WSDL* para este serviço, como mostra a Fig. 5.2.

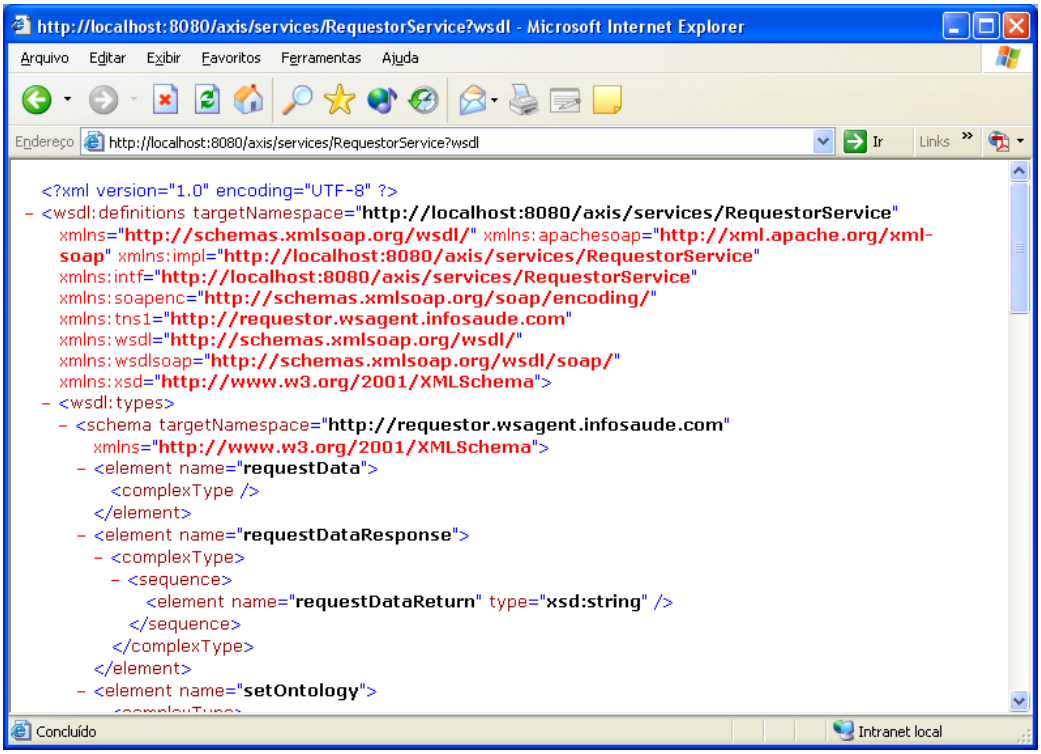


FIGURA 5.2 – *WSDL* do serviço.

O elemento `wsdl:definitions` traz, no atributo `targetNamespace` (segunda linha da estrutura *WSDL* mostrada na Fig. 5.2), o endereço a partir do qual o *Web Service* poderá ser invocado. Neste caso, o endereço é `http://localhost:8080/axis/services/RequestorService`.

É possível disponibilizar, como serviços, desde programas simples até componentes Java chamados Enterprise Java Beans (EJBs), bastando fazer as configurações necessárias para cada caso. Para isso, basta seguir as orientações constantes na documentação das ferramentas.

5.3 CAMADA DE ONTOLOGIA

Os *Web Services* são "alimentados" com arquivos *XML* que representam Ontologias. Durante a execução de um Agente, a classe `OntologyReader` carrega o conteúdo do arquivo XML para uma estrutura em memória, possibilitando que as demais classes façam acesso a suas informações sem a necessidade de buscá-las em disco.

Quando é solicitada a atualização da Ontologia, a classe `OntologyReader` recebe o novo arquivo de Ontologia (`newOntology`) e realiza a sua gravação sobre o arquivo já existente (cujo endereço está em `ontologyPath`), como é mostrado a seguir.

```
XMLUtils.DocumentToStream(newOntology,
                      new FileOutputStream(ontologyPath.toString()));
```

Foi definido um local padrão para armazenamento dos arquivos de Ontologia: a pasta `ws\ontology`, que deve ser criada dentro do diretório referente ao tipo de servidor *JBoss* utilizado. Neste diretório devem ficar armazenados os arquivos de Ontologia dos Agentes instalados no respectivo servidor, bem como o arquivo `OntologySchema.xsd`, que contém o *XML Schema* a partir do qual as Ontologias são validadas.

Um pequeno exemplo de arquivo de Ontologia, que é válido segundo o *XML Schema* mostrado no Capítulo 4 (ver seção 4.5), é mostrado a seguir:

```
<ontology>
  <class name=”paciente”>
    <property name=”codigo”>
      <field>PACIENTE.codigo</field>
      <type>string</type>
    </property>
    <property name=”nome”>
      <field>PACIENTE.nome</field>
      <type>string</type>
      <rule>splitName</rule>
    </property>
  </class>
  <class name=”leito”>
    <property name=”codigo”>
      <field>LEITO.cd_quarto</field>
      <type>integer</type>
    </property>
    <property name=”numero”>
      <field>LEITO.leito</field>
      <type>string</type>
    </property>
</ontology>
```

Este arquivo define duas classes: `paciente` e `leito`. Para cada `paciente`, é preciso saber onde ficam armazenados seu código (propriedade `codigo`) e seu nome (propriedade `nome`), que podem ser obtidos nos campos `codigo` e `nome` da tabela `PACIENTE`; o nome tem uma regra associada (`splitName`), que será executada sempre que esse dado for manipulado. Para `leito`, é preciso conhecer o código (propriedade `codigo`) e o número do leito (propriedade `numero`), que podem ser obtidos nos campos `codigo` e `leito` da tabela `LEITO`.

Como sempre haverá um arquivo de Ontologia para um Agente do tipo *Requestor* e outro arquivo de Ontologia para um Agente do tipo *Provider*, uma regra adicional para a criação destes arquivos é que eles devem possuir a mesma estrutura. Isto significa que devem possuir as mesmas classes, nomeadas de forma idêntica, com as mesmas propriedades (e estas também devem manter o mesmo nome). Apenas podem variar os tipos de dados, regras e

condições para cláusula `where`, e o Agente *Provider* pode omitir algumas propriedades caso não faça sentido buscar alguma informação.

Como exemplo, suponhamos que a Ontologia mostrada anteriormente é utilizada pelo Agente *Requestor*. O Agente *Provider* poderia ter sua Ontologia definida como no trecho a seguir.

```
<ontology>
  <class name=”paciente”>
    <property name=”nome”>
      <field>PACIENTE.nome_paciente</field>
      <type>string</type>
      <where>PAC_HOSPITAL.cod_paciente=PACIENTE.cod_paciente</where>
    </property>
  </class>
  <class name=”leito”>
    <property name=”numero”>
      <field>QUARTO_LEITO.leito</field>
      <type>string</type>
      <where>PAC_HOSPITAL.leito=QUARTO_LEITO.leito</where>
    </property>
</ontology>
```

Este arquivo define as mesmas classes `paciente` e `leito`. Para um `paciente`, já não é necessário saber onde está armazenado seu código (propriedade `codigo`), uma vez que este não será enviado para o Agente *Requestor*; apenas é necessário saber onde obter seu nome (propriedade `nome`), que fica armazenado no campo `nome_paciente` da tabela `PACIENTE`. Esta propriedade possui uma cláusula `where` associada, definindo que devem ser selecionados os registros onde o código associado ao paciente (`PACIENTE.cod_paciente`) estejam presentes na tabela de pacientes internados (`PAC_HOSPITAL.cod_paciente`). Da mesma forma, para `leito`: não é necessário obter seu código e há uma cláusula `where` associada, definindo que devem ser selecionados os registros onde o leito do paciente (`PAC_HOSPITAL.leito`) seja igual ao código de leito cadastrado (`QUARTO_LEITO.leito`).

A conversão do formato XML da Ontologia para um conjunto de registros selecionados na base de dados é feito pela classe adaptadora `OntologyToResultSet`, que varre os elementos da Ontologia e armazena uma lista de tabelas, uma lista de campos e uma

lista de filtros para a cláusula `where`, que serão utilizadas pela camada de persitência para construir a consulta *SQL*. O trecho de código a seguir exemplifica o que é feito:

```
ArrayList fields = new ArrayList();
ArrayList tables = new ArrayList();
ArrayList filters = new ArrayList();
Iterator classIterator = ontology.getRootElement()
                                    .getChildren("class").iterator();

while (classIterator.hasNext()) {
    Element ontologyClassElement = (Element)classIterator.next();
    Iterator propertyIterator = ontologyClassElement
                                  .getChildren("property").iterator();

    while (propertyIterator.hasNext()) {
        Element ontologyPropertyElement = (Element)propertyIterator.next();
        String field = ontologyPropertyElement
                                       .getChild("field").getText();
        fields.add(field);
        String table = getTableName(field);

        if (!tables.contains(table)) {
             tables.add(table);
        }

        Iterator filterIterator = ontologyPropertyElement
                                    .getChildren("where").iterator();

        while (filterIterator.hasNext()) {
             Element ontologyFilterElement = (Element)filterIterator.next();
             filters.add(ontologyFilterElement.getText());
        }
    }
}
```

Os dados obtidos a partir da Ontologia são transformados novamente para XML, num formato que permitirá sua posterior utilização pelo Agente *Requestor*. Esta transformação é feita pela classe adaptadora `ResultSetToXML` que, para cada registro do conjunto de dados, varre a Ontologia e constrói o arquivo XML tratando aspectos como o tipo dos dados e as regras de negócios que porventura devam ser executadas esses dados.

Cada tipo de dados que pode ser indicado na Ontologia é convertido para um tipo correspondente na linguagem Java:

- `string` corresponde a `java.lang.String`;
- `integer` corresponde a `java.lang.Integer`;

- `long` corresponde a `java.lang.Long`;
- `float` corresponde a `java.lang.Float`;
- `double` corresponde a `java.lang.Double`;
- `date` corresponde a `java.sql.Date`.

Cada regra indicada na Ontologia é instanciada de forma indireta e o método correspondente deve estar implementado na classe `BusinessRules`, respeitando as seguintes normas:

- o método deve ser implementado com nome exatamente igual àquele indicado na Ontologia, inclusive quanto a letras maiúsculas e minúsculas;
- o método deve possuir tipo de retorno correspondente ao tipo de dado indicado na Ontologia;
- o método deve receber como parâmetro o conteúdo de todas as *tags* `<field>`, que correspondem aos dados a serem manipulados; deve haver um parâmetro para cada *tag*, declarado com tipo correspondente ao tipo de dado indicado na Ontologia.

O trecho de código a seguir exemplifica a instanciação e invocação de um método de forma indireta, caso haja apenas um parâmetro a ser manipulado:

```
rules = BusinessRules.getIntance();
(…)
Method ruleMethod = rules.getClass().getMethod(ruleMethodValue,
                                new Class[]{convertedData.getClass()});
ruleMethod.invoke(rules, new Object[]{convertedData});
```

No código, `ruleMethodValue` corresponde ao nome do método que foi obtido na Ontologia e `convertedData` corresponde ao dado no formato correspondente ao que foi indicado também na Ontologia.

O conteúdo XML gerado é enviado pelo Agente *Provider* para o Agente *Requestor*, que realiza seu processamento através da classe `XMLToData` (que também permite a execução de regras de negócio sobre os dados a serem utilizados para atualização). Com auxílio da Ontologia, o Agente tenta realizar a atualização; caso não encontre algum registro, realiza a inserção dos dados.

5.4 CAMADA DE PERSISTÊNCIA

Para apoiar os adaptadores baseados em *XML*, a camada de persitência foi desenvolvida utilizando *JDBC*. Para cada Agente, esta camada consiste da interface `DB` e de outra classe que a implemente; esta última deve ser indicada na interface `DB` como segue abaixo, para que posa ser instanciada:

```
public static final String DB_CLASS_NAME =
              "com.infosaude.wsagent.requestor.persistence.OracleDB";
```

No exemplo, A classe `OracleDB` implementa a interface `DB` e será instanciada para operar com a base de dados do Agente *Requestor*.

A classe indicada na interface `DB` é instanciada da mesma forma em todas as classes adaptadoras que fazem acesso a bases de dados:

```
Method instanceMethod = Class.forName(DB.DB_CLASS_NAME)
                                        .getMethod("getInstance", null);
DB dataBase = (DB)instanceMethod.invoke(dataBase, null);
```

Em tempo de execução, é consultado o atributo `DB_CLASS_NAME` da interface `DB`; o método `getInstance` desta classe é obtido em `instanceMethod` e então invocado, guardando a instância da classe em `dataBase`.

A classe que implementa a interface `DB` deve informar, para poder realizar a conexão, o driver do banco de dados utilizado, bem como usuário e senha de acesso à base de dados. Isto é informado no método `getConnection`. O trecho a seguir exemplifica o que foi dito para uma base *SQL Server*.

```
Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
String connURL = "jdbc:microsoft:sqlserver://localhost:1433;User=sa;
                          Password=sauser;DatabaseName=TESTE_LEGADO";
Connection conn = DriverManager.getConnection(connURL);
```

5.5 COMO INSTANCIAR O *FRAMEWORK*

O processo de instanciação é constituído das seguintes etapas:

1) renomear as classes `Requestor` e `Provider` de forma a indicar o processo a que estão relacionadas; isto facilita sua identificação e evita conflitos de nomes quando os Agentes forem disponibilizados no servidor de aplicações (o nome do processo deve ser incuído antes das palavras “requestor” e “provider”). Este nome também deve ser alterado na classe `OntologyReader`, que guarda o caminho para obtenção do arquivo de Ontologia;
2) definir a Ontologia de interoperabilidade para cada Agente, com base no *XML Schema* apresentado no Capítulo 4 (ver seção 4.5). Cada arquivo deve receber o nome do Agente ao qual está relacionado, mais a palavra “Ontology”;
3) codificar as regras de negócios dos Agentes (classes `BusinessRules`), de forma coerente com os arquivos de Ontologia e as regras informadas na seção 5.3;
4) implementar a interface `DB` conforme as instruções fornecidas na seção 5.4. É boa prática que se crie uma classe com nome referente ao tipo de banco de dados utilizado (por exemplo, `SQLServerDB` ou `OracleDB`).

Após, os Agentes podem ser disponibilizados. Uma boa prática é que o(s) arquivo(s) de extensão `jar` que guardam as classes dos Agentes recebam um nome que facilite sua identificação (por exemplo, o nome do processo relacionado). O código que realiza a chamada do Agente *Requestor* deve ser incluído na aplicação que disparará a atualização dos dados, no ponto em que for conveniente, segundo as orientações da seção 5.2.

O processo relacionado a pacientes e leitos no sistema Clínico Assistencial, com a sugestão da empresa InfoSaúde, foi definido como o primeiro caso a ser estudado e serviu como base para a instanciação dos Agentes. Estes Agentes foram nomeados `PacienteRequestor` e `PacienteProvider`.

Foi realizada a análise da estrutura de informações contida em cada base de dados, a fim de selecionar as tabelas e campos necessários para integração. Com base nisto, foram definidos os arquivos de Ontologia.

A seguir, é mostrado um trecho do arquivo `PacienteRequestorOntology.xml`, que define a Ontologia de interoperabilidade para o Agente `PacienteRequestor`.

```
<ontology>
  <class name=”paciente”>
    <property name=”codigo”>
      <field>CA_PACIENTE.codigo</field>
      <type>string</type>
    </property>
    <property name=”nome”>
      <field>CA_PACIENTE.nome</field>
      <type>string</type>
    </property>
    <property name=”nascimento”>
      <field>CA_PACIENTE.datanascimento</field>
      <type>date</type>
    </property>
    <property name=”sexo”>
      <field>CA_PACIENTE.sexo</field>
      <type>string</type>
    </property>
    <property name=”cor”>
      <field>CA_PACIENTE.cor</field>
      <type>string</type>
    </property>
    (…)
    <property name=”endereco_residencial”>
```

```
      <field>CA_PACIENTE_EFETIVO.endereco_residencial</field>
      <type>string</type>
    </property>
    <property name=”telefone”>
      <field>CA_PACIENTE_EFETIVO.telefone</field>
      <type>string</type>
    </property>
    (…)
    <property name=”prontuario”>
      <field>CA_PACIENTE_HOSPITAL.prontuario</field>
      <type>int</type>
      <rule>defineProntuario</rule>
    </property>
  </class>
  (…)
</ontology>
```

A seguir, é mostrado um trecho do arquivo `PacienteProviderOntology.xml`, que define a Ontologia de interoperabilidade para o Agente `PacienteProvider`.

```
<ontology>
  <class name=”paciente”>
    <property name=”nome”>
      <field>PACIENTE.nm_paciente</field>
      <type>string</type>
    </property>
    <property name=”nascimento”>
      <field>PACIENTE.dt_nascimento</field>
      <type>date</type>
    </property>
    <property name=”sexo”>
      <field>PACIENTE.sexo</field>
      <type>string</type>
    </property>
    <property name=”cor”>
      <field>COR_PELE.sc_cor</field>
      <type>string</type>
      <where>COR_PELE.cd_cor=PACIENTE.cd_cor</where>
    </property>
    (…)
    <property name=”endereco_residencial”>
      <field>PACIENTE.nm_logradouro</field>
      <field>PACIENTE.nr_logradouro</field>
      <field>PACIENTE.compl_logradouro</field>
      <type>string</type>
      <rule>defineEnderecoResidencial</rule>
    </property>
    <property name=”telefone”>
      <field>PACIENTE.nr_ddd_fone</field>
      <field>PACIENTE.nr_fone</field>
      <type>string</type>
      <rule>defineTelefone</rule>
    </property>
    (…)
    <property name=”prontuario”>
      <field>H_PACIENTE.nr_arquivo_fia</field>
      <field>H_PACIENTE.nr_arquivo_baa</field>
```

```
      <type>int</type>
    </property>
  </class>
  (…)
</ontology>
```

Após a definição dos arquivos de Ontologia, foram implementadas as regras de negócio dos Agentes. Por exemplo, para o Agente *Provider* foi implementada a regra `defineTelefone`, associada à propriedade `telefone` da classe `paciente`. Esta regra agrupa as informações sobre DDD e número do telefone, para que possam ser enviadas ao Agente *Requestor* como uma única informação. A seguir, é mostrada a implementação da regra.

```
public String defineTelefone(String ddd, String numero) {
   StringBuffer buffer = new StringBuffer(ddd);
   buffer.append(" ");
   buffer.append(numero);
   return buffer.toString();
}
```

A etapa seguinte consistiu da implementação das classes `OracleDB` para os Agentes, as quais implementam a interface `DB`. Seguindo as orientações da seção 5.4, em cada uma das classes o driver *Oracle* foi informado no método `getConnection`, bem como o usuário e a senha da base de dados relacionada. Um dos demais métodos a serem codificados é `executeSelect`, mostrado a seguir.

```
Connection conn = null;
Statement stmt = null;

(…)

public ResultSet executeSelect(List fields, List tables, List links)
                                                    throws SQLException {
   StringBuffer selectQuery = new StringBuffer("SELECT ");
   Iterator elementIterator = fields.iterator();

   while (elementIterator.hasNext()) {
      selectQuery.append((String)elementIterator.next());

      if (elementIterator.hasNext()) {
         selectQuery.append(", ");
      }
   }
```

```
        selectQuery.append(" FROM ");
        elementIterator = tables.iterator();

        while (elementIterator.hasNext()) {
           selectQuery.append((String)elementIterator.next());

           if (elementIterator.hasNext()) {
              selectQuery.append(", ");
           }
        }

        if (!links.isEmpty()) {
           selectQuery.append(" WHERE ");
           elementIterator = links.iterator();

           while (elementIterator.hasNext()) {
              selectQuery.append((String)elementIterator.next());

              if (elementIterator.hasNext()) {
                 selectQuery.append(" AND ");
              }
           }
        }

        return stmt.executeQuery(selectQuery.toString());
     }
```

Por fim, os Agentes foram disponibilizados no servidor de aplicações e foi escolhida a classe `ConsultaPaciente` do sistema Clínico Assistencial (método `getInstance`) para incluir a chamada do Agente *Requestor*. Assim, sempre que for realizada uma consulta de paciente, o processo de atualização será iniciado.

Para criar uma instância da solução que aborde outro processo como, por exemplo, cadastro de Profissional no Sistema Clínico Assitencial, é possível aproveitar a maior parte do código já implementado. Isto foi verificado, sendo necessárias as seguintes alterações:

- renomear as classes `Requestor` e `Provider` para `ProfissionalRequestor` e `ProfissionalProvider`;
- criar os arquivos de Ontologia que referentes aos Agentes, nomeando os arquivos `ProfissionalRequestorOntology.xml` e `ProfissionalProviderOntology.xml`;
- alterar a classe `OntologyReader` de cada Agente para que localize o arquivo de Ontologia correto;

- disponibilizar os Agentes no servidor de aplicações e incluir a chamada ao Agente *Requestor* na classe `ConsultaProfissional` (método `getInstance`), de forma a iniciar o processo de atualização sempre que houver consulta de profissionais.

Neste caso, as bases de dados utilizadas são as mesmas. Caso fosse necessário utilizar um outro tipo de banco de dados, bastaria utilizar uma classe já implementada ou criar uma nova classe para manipulá-lo de acordo com a sintaxe adequada. Sempre é necessário informar o *driver* a ser utilizado para conexão com a base de dados, bem como usuário e senha de acesso.

Alguns testes foram realizados no HMV. Para isto, foi disponibilizada a versão 3.2.x do servidor de aplicações *JBoss*, disponibilizando uma versão do sistema Clínico Assistencial e também da base legada, em uma máquina preparada para testes. Neste ambiente de testes é utilizado banco de dados *Oracle*, com uma base para a aplicação Clínico Assistencial e outra para o sistema corporativo (base legada).

Outros testes, feitos fora do HMV, incluíram bases de dados *SQL Server* (*Oracle* e *SQL Server* foram indicados pela InfoSaúde bancos de dados bastante utilizados por seus sistemas).

Há mais processos que devem ser abordados e que ainda necessitarão de testes.

O trabalho foi aceito para publicação na *OOPSLA* 2004 (*19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*), ocorrida em outubro. Uma cópia do trabalho encontra-se em anexo.

O *WSAgent* não pode ser considerado totalmente pronto. Ele necessita ser incrementado, devido ao interesse da InfoSaúde em torná-lo uma ferramenta comercial.

6 CONCLUSÃO

Este trabalho apresentou o *WSAgent*, uma proposta de solução para integração de bases de dados heterogêneas. O problema de integração é freqüentemente encontrado no domínio da saúde, devido ao grande número de sistemas já existentes e à demanda para criação de novos sistemas.

Diversas tecnologias também foram apresentadas: *Frameworks*, *Design Patterns*, Ontologias, Agentes de *Software* e *Web Services*.

Frameworks podem ser vistos como um conjunto de classes que constitui um *design* abstrato para soluções dentro de um mesmo contexto (por exemplo, a área da saúde). *Design Patterns* constituem soluções para problemas recorrentes no desenvolvimento de *software*, reconhecidas como boas práticas e bem documentadas. Estas tecnologias são utilizadas em conjunto, visando obter maior flexibilidade e reusabilidade.

Ontologias traduzem um vocabulário comum, expressam o conhecimento sobre um determinado tema. Agentes de *Software* fazem uso dessa representação de conhecimento para realizar suas tarefas, obtendo uma visão compatibilizada das informações que processam.

Web Services podem ser vistos como Componentes de *Software* independentes de linguagem de programação ou plataforma, que são disponibilizados na *Web* (ou em uma *Intranet*) através de uma série de tecnologias padronizadas. A base dessas tecnologias é a linguagem *XML*, que proporciona interoperabilidade.

O *WSAgent* visa aproveitar os pontos positivos das tecnologias apresentadas e vem como uma proposta diferenciada, uma vez que:

- Os sistemas a serem integrados não necessitam ter seu código alterado a cada mudança no ambiente;
- *Web Services* permitem total distribuição, não precisando estar instalados em um mesmo servidor de aplicações ou em uma mesma máquina;
- Há uma grande interoperabilidade, promovida pelo uso de tecnologias baseadas em *XML*;
- Sua arquitetura permite que apenas a camada de Ontologia, que contém o conhecimento sobre os dados que devem ser atualizados, seja modificada em caso de alguma mudança no ambiente.

No entanto, o *WSAgent* ainda precisa amadurecer para tornar-se uma solução completa.

Uma das maiores dificuldades enfrentadas no desenvolvimento do trabalho foi a grande quantidade de material a ser estudado para obter a compreensão necessária das tecnologias e ferramentas disponíveis. Isto tomou uma parte considerável do tempo disponível para realização do trabalho.

Especificamente no caso dos *Web Services*, há uma grande diversidade de abordagens e ferramentas. A partir de 2002, quando as tecnologias de *Web Services* começaram a se consolidar, muitos trabalhos começaram a ser desenvolvidos e outros, já em andamento, tomaram mais força. Muitos desses trabalhos seguiram rumos diferentes. Os livros disponíveis, em geral, não abordam o assunto de forma atualizada. A maior dificuldade, neste sentido, é que enquanto se está consolidando uma forma de contornar alguma falha ou deficiência da ferramenta utilizada para desenvolvimento, surge uma outra versão ou outra ferramenta que resolve o problema de forma total ou parcial. E, muitas vezes, há uma certa mudança na utilização dessas versões.

O desenvolvimento do trabalho não pára por aqui. A seguir serão abordados alguns pontos que necessitam de prosseguimento.

6.1 TRABALHOS FUTUROS

A questão da segurança, sem dúvida, é uma grande necessidade para uma aplicação deste tipo. Tolerância a falhas e controle de transações são outros aspectos bastante relevantes que necessitam ser tratados.

Estas e outras questões, como o gerenciamento dos Agentes no ambiente e o processo de geração/configuração precisam ser desenvolvidos ou melhorados para que o *WSAgent* possa realmente ser aplicado comercialmente.

A seguir, são detalhados os principais pontos que devem ser tratados para que a solução se torne uma ferramenta comercial:

- segurança – prevenção quanto a possíveis ataques (negação do serviço, interceptação e manipulação de mensagens, falsificação de requisições, falsificação de respostas, tentativas de acesso ao sistema de arquivos/base de dados). Algumas soluções possíveis são a implementação de mecanismos de segurança nos Agentes, a configuração de mecanismos do *Axis* e autilização de recursos de segurança no ambiente onde os Agentes estão instalados;
- controle de transações – garantia de integridade dos dados (sessões *EJB, two phase commit*) e utilização de *logs* de recuperação pós-falha;
- recursos avançados – notificação de falhas (via *e-mail*, por exemplo) e gerenciamento remoto dos *Web Services* (ativar/desativar os Agentes, atualizar a Ontologia, atualizar os Agentes);

- melhorias no gerador de Ontologias – disponibilização da geração das classes dos Agentes e adição de recursos de gerenciamento remoto.

A empresa InfoSaúde demonstrou interesse na continuidade do desenvolvimento do trabalho. Assim, os pontos que hoje deixam a desejar serão contemplados e a solução, em especial o *Framelet*, terá um grande amadurecimento.

Há também o interesse de obter mais publicações sobre o trabalho à medida que este for evoluindo. Um artigo está em andamento, visando a publicação em alguma boa revista da área de Engenharia de *Software*.

O desenvolvimento do *WSAgent* foi complexo e trabalhoso em vários aspectos, mas também bastante gratificante, devido à possibilidade de real aplicação da solução. Sem dúvidas, o período de desenvolvimento da dissertação foi de grande aprendizado.

REFERÊNCIAS

ALEXANDER, Christopher et al. A pattern language: towns, buildings, construction. New York: Oxford University Press, 1977.

ALUR, Deepak; CRUPI, John; MALKS, Dan. **Core J2EE patterns**: as melhores práticas e estratégias de design. Tradução de Altair Dias Caldas de Moraes, Cláudio Belleza Dias e Guilherme Dias Caldas de Moraes. Rio de Janeiro: Campus, 2002. 406 p.

AMOR, Mercedes et al. Combining software components and mobile agents. In: INTERNATIONAL WORKSHOP ON ENGINEERING SOCIETIES IN THE AGENTS' WORLD, 1., Aug. 2000. Berlin. **Lecture notes in artificial intelligence**. Berlin: Springer-Verlag, v. 1972, 2000. p. 129-142.

APPLETON, Brad. **Patterns and software**: essential concepts and terminology. Disponível em: <http://www.cmcrossroads.com/bradapp/docs/patterns-intro.pdf>. Acesso em: 14 mai. 2004.

BENCH-CAPON, Trevor J. M.; VISSER, Pepijn R. S. Ontologies in legal information systems: the need for explicit specifications of domain conceptualisations. In: INTERNATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE AND LAW, 6., 1997. Melbourne. **Proceedings**… New York: ACM, 1997. p. 132-141.

BERNERS-LEE, Tim; HENDLER, James; LASSILA, Ora. The Semantic Web. **Scientific American**, May 2001. Disponível em: <http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21>. Acesso em: 14 mai. 2004.

BEZEK, Andraz; GAMS, Matjaz. Agent-oriented software engineering: a comparison of agent and non-agent version of a cluster server. In: INTERNATIONAL JOINT CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, 2., July 2003. Melbourne. **Proceedings**… New York: ACM, 2003. p. 930-931.

BIRD, Linda J.; GOODCHILD, Andrew; BEALE, T. Integrating health care information using XML-based metadata. In: NATIONAL HEALTH INFORMATICS CONFERENCE, 8., Sep. 2000. Adelaide. **Proceedings**… Disponível em: <http://titanium.dstc.edu.au/papers/hic2000.pdf>. Acesso em: 14 mai. 2004.

BIRD, Linda J.; GOODCHILD, Andrew; HEARD, Sam. Importing clinical data into electronic health records. In: NATIONAL HEALTH INFORMATICS CONFERENCE, 10., Aug. 2002. Melbourne. **Proceedings**… Disponível em: <http://titanium.dstc.edu.au/papers/hic2002.pdf>. Acesso em: 14 mai. 2004.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **The Unified Modeling Language user guide**. Reading: Addison-Wesley, 1999. 482 p.

BRADSHAW, Jeffrey M. (Ed.). **Software agents**. Cambridge: MIT, 1997. 480 p.

BURGUN, A. et al. Issues in the design of medical ontologies used for knowledge sharing. **Journal of Medical Systems**, New York, v. 25, n. 2, Apr. 2001. p. 95-108.

BUSSLER, Christoph; FENSEL, Dieter; MAEDCHE, Alexander. A conceptual architecture for semantic web enabled web services. **ACM SIGMOD Record**, New York, v. 31, n. 4, Dec. 2002. p. 24-29.

CAMPOS, Fernanda A. C.; SANTOS, Neide; BRAGA, Regina M. M. Ontologias para o domínio da educação mediada pela web. In: SIMPÓSIO BRASILEIRO DE INFORMÁTICA NA EDUCAÇÃO, 13., Nov. 2002. São Leopoldo. **Anais**... São Leopoldo: UNISINOS, 2002. p. 612-615.

CASATI, Fabio et al. Business-oriented management of web services. **Communications of the ACM**, New York, v. 46, n. 10, Oct. 2003. p. 55-60.

CASTANO, Silvana; ANTONELLIS, Valeria de. A discovery-based approach to database ontology design. **Distributed and Parallel Databases**, Hingham, v. 7, n. 1, Jan. 1999. p. 67-98.

CAULDWELL, Patrick et al. **Professional XML web services**. Chicago: Wrox, 2001. 803 p.

CHANDRASEKARAN, B. What are ontologies, and why do we need them? **IEEE Intelligent Systems**, Piscataway, v. 14, n. 1, Jan. 1999. p. 20-26.

CHUNG, Jen-Yao; LIN, Kwei-Jay; MATHIEU, Richard G. Web services computing: advancing software interoperability. **IEEE Computer**, v. 36, n. 10, Oct. 2003. p. 35-37.

COEN, Michael. **The SodaBot agent**. [199-]. Disponível em: <http://www.ai.mit.edu/people/sodabot/slideshow/total/P001.html>. Acesso em: 14 mai. 2004.

COYLE, Frank P. **XML, web services, and the data revolution**. Boston: Addison-Wesley, 2002. 356 p.

CRESPO C. S. PINTO, Sérgio; LUCENA, Carlos José Pereira de; STAA, Arndt Von. **FrameConstructor**: uma ferramenta para suporte à construção sistemática de frameworks. Rio de Janeiro: PUC, 1998. 45 p. (Monografias em Ciências da Computação, 42/98).

CRESPO C. S. PINTO, Sérgio. **Composição em WebFrameworks.** Rio de Janeiro, 2000. 150 f. Tese (Doutorado) – Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

CUESTA-MORALES, Pedro et al. Agent technologies at work. **UPGRADE**, [s.l.], v. 5, n. 4, Aug. 2004. p. 3-5. Disponível em: <http://www.upgrade-cepis.org/issues/2004/4/upgrade-vol-V-4.html>. Acesso em: 14 mai. 2004.

CUI, Z.; TAMMA V. A. M.; BELLIFEMINE, F. Ontology management in enterprises. **BT Technology Journal**, v. 17, n. 4, Oct. 1999. p.98-107.

CURBERA, F. et al. The next step in web services. **Communications of the ACM**, New York, v. 46, n. 10, Oct. 2003. p. 29-34.

DAMIANI, E.; DE CAPITANI DI VIMARCATI, S.; SAMARATI, P. Towards securing XML web services. In: ACM WORKSHOP ON XML SECURITY, Nov. 2002. **Proceedings**… New York: ACM, 2002. p. 90-96.

DEVEDZICK, V. Ontologies: borrowing from software patterns. **Intelligence**, Fall 1999. p. 14-24.

DICKINSON, I.; WOOLDRIDGE, M. Towards practical reasoning agents for the semantic web. In: INTERNATIONAL JOINT CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, 2., July 2003. Melbourne. **Proceedings**… New York: ACM, 2003. p. 827-834.

DUELL, M. Catalog of non-software examples of design patterns. **Object Magazine**, v. 7, n. 5, July 1997. p. 52-57. Disponível em: <http://www2.ing.puc.cl/~jnavon/IIC2142/patexamples.htm>. Acesso em: 14 mai. 2004.

DURHAM, J. **Tracing the roots of components from OOP through WS**. Apr. 2001. Disponível em: <http://www-106.ibm.com/developerworks/library/co-tmline/index.html>. Acesso em: 14 mai. 2004.

ELLINGSEN, G. **Integration strategies in hospitals**. 2001. Disponível em: <http://www.idi.ntnu.no/emner/dif8914/essays/gunnar-essay.pdf>. Acesso em: 5 dez. 2004.

ESTOMBELO-MONTESCO, C. A.; MOREIRA, D. A. UCL: uma linguagem de comunicação para agentes de software baseada em ontologias. In: WORKSHOP EM TECNOLOGIA DA INFORMAÇÃO E DA LINGUAGEM HUMANA, 1., Out. 2003. São Carlos. **Anais**… 2003. Disponível em: <http://www.nilc.icmc.usp.br/til2003/poster/ estombelo_moreira_10.pdf >. Acesso em: 14 mai. 2004.

FALBO, R. A.; GUIZZARDI, G.; DUARTE, K. C. An ontological approach to domain engineering. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING, 14., July 2002. Ischia. **Proceedings**… New York: ACM, 2002. p. 351-358.

FAYAD, Mohamed E.; SCHMIDT, Douglas C.; JOHNSON, Ralph E. **Building application frameworks**: object-oriented foundations of framework design. New York: John Wiley & Sons, 1999. 664 p.

FAYAD, Mohamed E.; SCHMIDT, Douglas C.; JOHNSON, Ralph E. **Implementing application frameworks**: object-oriented frameworks at work. New York: John Wiley & Sons, 1999. 729 p.

FAYAD, Mohamed E.; JOHNSON, Ralph E. **Domain-specific application frameworks**: frameworks experience by industry. New York: John Wiley & Sons, 1999. 681 p.

FELICÍSSIMO, C. H. et al. Geração de ontologias subsidiada pela engenharia de requisitos. In: Workshop em Engenharia de Requisitos, 6., Nov. 2003. Piracicaba. **Anais**... 2003. p. 255-269.

FERREIRA, Steferson L. C.; GIRARDI, Rosario. Arquiteturas de software baseadas em agentes: do nível global ao detalhado. **Revista Eletrônica de Iniciação Científica**, v. 2, n. 2, jun. 2002. Disponível em: <http://www.sbc.org.br/reic/edicoes/2002e2/>. Acesso em: 14 mai. 2004.

FERRIS, C.; FARRELL, J. What are web services? **Communications of the ACM**, New York, v. 46, n. 6, Jun. 2003. p. 31.

FOSTER, I.; GROSSMAN, R. L. Data integration in a bandwidth-rich world. **Communications of the ACM**, New York, v. 46, n. 11, Nov. 2003. p. 50-57.

FRANKLIN, Stan; GRAESSER, Art. Is it an agent, or just a program?: A taxonomy for autonomous agents. In: INTERNATIONAL WORKSHOP ON AGENT THEORIES, ARCHITECTURES, AND LANGUAGES, 3., 1996. Budapest. **Lecture notes in computer science**. Budapest: Springer-Verlag, v. 1193, 1997. p. 21-35. Disponível em: <http://www.msci.memphis.edu/~franklin/AgentProg.html>. Acesso em: 14 mai. 2004.

FREITAS, Frederico L. G. **Sistemas multiagentes cognitivos para a recuperação, classificação e extração integradas de informação da web**. Florianópolis, 2002. Tese (Doutorado) – Universidade Federal de Santa Catarina.

GAMMA, Erich. **Design patterns**: elements of reusable object-oriented software. Reading: Addison-Wesley, 1995. 395 p.

GARCIA, Alessandro et al. Engineering multi-agent systems with aspects and patterns. **Journal of the Brazilian Computer Society**, [s.l.], v.8, n.1, Aug. 2002. p. 57-72.

GEER, D. Taking steps to secure web services. **IEEE Computer**, Oct. 2003. p. 14-16.

GIBBINS, N.; HARRIS, S.; SHADBOLT, N. Agent-based Semantic Web Services. In: INTERNATIONAL WORLD WIDE WEB CONFERENCE, 12., May 2003. Budapest. **Proceedings**… New York: ACM, 2003. p. 710-717.

GLUSHKO, R. J.; TENENBAUM, J. M.; MELTZER, B. An XML framework for agent-based e-commerce. **Communications of the ACM**, New York, v. 42, n. 3, Mar. 1999. p. 106-114.

GOÑI, J. L. et al. Geração de ontologias usando Protègè-2000 para reuso de conteúdos educacionais numa arquitetura multiagente. In: SIMPÓSIO BRASILEIRO DE INFORMÁTICA NA EDUCAÇÃO, 13., Nov. 2002. São Leopoldo. **Anais**... São Leopoldo: UNISINOS, 2002. p. 571-574.

GORDON, A. D.; PUCELLA, R. Validating a web service security abstraction by typing. In: ACM WORKSHOP ON XML SECURITY, Nov. 2002. Washington. **Proceedings**… New York: ACM, 2002. p. 18-29.

GRAHAM, Steve. **Building web services with Java**: making sense of XML, SOAP, WSDL, and UDDI. Indianapolis: Sams, 2002. 581 p.

GROSOF, B. N.; POON, T. C. SweetDeal: Representing agent contracts with exceptions using XML rules, ontologies, and process descriptions. In: INTERNATIONAL WORLD WIDE WEB CONFERENCE, 12., May 2003. Budapest. **Proceedings**… New York: ACM, 2003. p. 340-349.

GRUBER, T. R. A translation approach to portable ontologies. **Knowledge Acquisition**, v. 2, n. 5, 1993. p.199-220.

GRÜNINGER, M.; ATEFI, K.; FOX, M. S. Ontologies to support process integration in enterprise engineering. **Computational & Mathematical Organization Theory**, n. 6, 2000. p. 381-394.

GUNZER, H. **Introduction to web services**. Mar. 2002. Disponível em: <http://www.brics.dk/~amoeller/web/intro_to_web_services_wp.pdf>. Acesso em: 14 mai. 2004.

HAKIMPOUR, F.; GEPPERT, A. Resolving semantic heterogeneity in schema integration: an ontology based approach. In: INTERNATIONAL CONFERENCE ON FORMAL ONTOLOGY IN INFORMATION SYSTEMS, 2., Oct. 2001. Ogunquit. **Proceedings**… New York: ACM, 2001. p. 297-308.

HANSEN, Roseli Persson et al. Web services: an architectural overview. In: SEMINAR ON ADVANCED RESEARCH IN ELECTRONIC BUSINESS, 1., Nov. 2002. Rio de Janeiro. **Proceedings**… 2002. p. 44-57.

HEALTH LEVEL SEVEN. **What is HL7?** Disponível em: <http://www.hl7.org/about/ about2.htm>. Acesso em: 14 mai. 2004.

HEALTH LEVEL SEVEN. **HL7 reference information model**. Jul 2004. Disponível em: <http://www.hl7.org/library/data-model/RIM/ C30204/rim.htm>. Acesso em: 05 dez. 2004.

HANSEN, Roseli Persson. **GlueScript**: uma linguagem específica de domínio para composição de web services. São Leopoldo, 2003. 89 fl. Dissertação (Mestrado) – Universidade do Vale do Rio dos Sinos, Centro de Ciências Exatas e Tecnológicas, Programa Interdisciplinar de Pós-Graduação em Computação Aplicada.

HILLSIDE.NET. **Patterns library**. Disponível em: <http://hillside.net/patterns/>. Acesso em: 14 mai. 2004.

HOODA, J. S.; DOGDU, E.; SUNDERRAMAN, R. Health Level-7 compliant clinical patient records system. In: ACM SYMPOSIUM ON APPLIED COMPUTING, 19., Mar. 2004. Nicosia. **Proceedings**… New York: ACM, 2004. p. 259-263.

INTERNATIONAL BUSINESS MACHINES. **The IBM Agent**. [19--]. Disponível em: <http://activist.gpl.ibm.com:81/WhitePaper/ptc2.htm>. Acesso em: 14 mai. 2004.

JASPER, R.; USCHOLD, M. A framework for understanding and classifying ontology applications. In: INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE, 16., July 1999. Stockholm. **Proceedings**… 1999. Disponível em: <http://sern.ucalgary.ca/KSI/KAW/KAW99/papers/Uschold2/final-ont-apn-fmk.pdf>. Acesso em: 14 mai. 2004.

JENNINGS, N. R.; WOOLDRIDGE, M. Software agents. **IEEE Review**, Jan. 1996. p. 17-20.

JENNINGS, N. R.; SYCARA, K.; WOOLDRIDGE, M. A roadmap of agent research and development. **Autonomous Agents and Multi-Agent Systems**, n. 1, 1998. p. 275-306.

JOHNSON, R. E. Reusing object-oriented design. Technical Report UIUCDCS 91-1696, University of Illinois, 1991.

JOHNSON, R. E.; Foote, B. Designing reusable classes. **Journal of Object-Oriented Programming**, v. 1, n. 2, June 1988. p. 22-35.

JONES, D.; BENCH-CAPON, T.; VISSER, P. Methodologies for ontology development. In: IFIP WORLD COMPUTER CONGRESS, 15., Aug. 1998. Budapest. **Proceedings of the IT&KNOWS Conference**. 1998. p. 62-75. Disponível em: <http://www.iet.com/Projects/RKF/SME/methodologies-for-ontology-development.pdf>. Acesso em: 12 abr. 2004.

KAY, A. Computer software. **Scientific American**, v. 3, n. 251, 1984. p. 53-59.

KIFER, M.; LAUSEN, G.; WU, J. **Logical foundations of object-oriented and frame-based languages**. Technical report, Jan. 1990. Disponível em: <citeseer.ist.psu.edu/kifer90logical.html>. Acesso em: 12 abr. 2004.

KIM, J. et al. Integrated multimedia medical data agent in e-health. In: PAN-SYDNEY AREA WORKSHOP ON VISUAL INFORMATION PROCESSING, Dec. 2001. Sydney. **Proceedings**… Sydney: Australian Computer Society, 2001. p. 11-15.

KNAPIK, Michael; JOHNSON, Jay. **Developing intelligent agents for distributed systems: exploring architecture, technologies and applications**. New York: McGraw-Hill, 1998.

KOTOK, A. **Business processes and web services**. Disponível em: <http://www.webservices.org/index.php/article/articleprint/859/-1/24/>. Acesso em: 14 mai. 2004.

KRAFT, R. Designing a distributed access control processor for network services on the web. In: ACM WORKSHOP ON XML SECURITY, Nov. 2002. Fairfax. **Proceedings**… New York: ACM, 2002. p. 36-52.

KRUTISCH, Richard; MEYER, Philipp; WIRSING, Martin. The AgentComponent Approach, combining agents and components. In: German Conference on Mutiagent System Technologies, 1., Sep. 2003. Erfurt. **Lecture notes in artificial intelligence**. Berlin: Springer Verlag, v. 2831, 2003. p. 1-12.

KUHN, K. A.; Giuse, D. A. From hospital information systems to health information systems: problems, challenges, perspectives. **Yearbook of Medical Informatics**, Stuttgart, 2001. p. 63-76.

LANGDON, C. S. The state of web services. **IEEE Computer**, July 2003. p. 93-94.

LAROIA, A. **Leveraging web services to connect the healthcare enterprise**. Disponível em: <http://www.ebizq.net/topics/healthcare/features/1546.html?page=1>. Acesso em: 05 dez. 2004.

LEVITT, J. From EDI to XML and UDDI: a brief history of web services. **InformationWeek**, Oct. 2001. Disponível em: <http://www.informationweek.com/ story/IWK20010928S0006/>. Acesso em: 14 mai. 2004.

LIMTHANMAPHON, B.; ZHANG, Y. Web service composition with case-based reasoning. In: AUSTRALASIAN DATABASE CONFERENCE ON DATABASE TECHNOLOGIES, 14., 2003. Adelaide. **Proceedings**… Darlinghurst: Australian Computer Society, v. 17, 2003. p. 201-208.

LIND, Jürgen. **Relating agent technology and component models**. 2001. Disponível em: http://www.agentlab.de/documents/Lind2001e.pdf >. Acesso em: 05 dez. 2004.

MAAMAR, Z. et al. Software agent-oriented frameworks for global query Processing. **Journal of Intelligent Information Systems**, n. 13, 1999. p. 235-259.

MAGLIO, P.; BARRET, R. Intermediaries personalize information streams. **Communications of the ACM**, New York, v. 43, n. 8, Aug. 2000. p. 96-101.

MARINESCU, Floyd. **EJB design patterns**: advanced patterns, processes, and idioms. New York: John Wiley & Sons, 2002. 259 p.

MARKIEWICZ, M.; LUCENA, C. J. **Understanding object-oriented framework engineering**. Rio de Janeiro: PUC, 2000. 11 p. (Monografias em Ciências da Computação, 38/00).

MARSHALL, C. C.; Shipman, F. M. Which semantic web? In: ACM CONFERENCE ON HYPERTEXT AND HYPERMEDIA, 14., Aug. 2003. Nottingham. **Proceedings**… New York: ACM, 2003. p. 57-66.

MATTSON, M. **Object-Oriented Frameworks**: a survey of methodological issues. 130 fl., 1996. Thesis (Licentiate) – Lund University, Department of Computer Science. Disponível em: <http://www.ipd.bth.se/michaelm/papers/Mattsson.Lic.thesis.pdf >. Acesso em: 14 mai. 2004.

MATTSON, M.; BOSCH, J.; FAYAD, M. Framework integration: problems, causes and solutions. **Communications of the ACM**, New York, v. 42, n. 10, Oct. 1999. p. 80-87.

MATTSON, M. **Evolution and composition of object-oriented frameworks**. 216 fl., 2000. Thesis (PhD) – University of Karlskrona/Ronneby, Department of Software Engineering and Computer Science. Disponível em: <http://www.ipd.bth.se/michaelm/papers/Michael.Mattsson. Ph.D.thesis.pdf >. Acesso em: 14 mai. 2004.

MEDJAHED, B.; BOUGUETTAYA, A.; ELMAGARMID, A. K. Composing web services on the semantic web. **TheVLDB Journal**, Secaucus, v. 2, n. 4, Nov. 2003. p. 333-351.

MOULIN, B.; CHAIB-DRAA, B. An Overview of Distributed Artificial Intelligence. In: G. O´HARE, M. P.; JENNINGS, N. R. (Ed.). **Foundations of Distributed Artificial Intelligence**. New York: John Wiley & Sons , 1996. p. 3–55.

MYKKÄNEN, J. et al. Integration models in health information systems: experiences from the PlugIT project. In: WORLD CONGRESS OF MEDICAL INFORMATICS, 11., Sep. 2004. San Francisco. **Proceedings**… Bethesda: AMIA, 2004. Disponível em: <http://www.medinfo2004.org/post/pdf/mykkanen-medinfo.pdf >. Acesso em: 05 dez. 2004.

NAEDELE, M. Standards for XML and web services security. **IEEE Computer**, Apr. 2003. p. 96-98.

NEWCOMER, Eric. **Understanding web services**: XML, SOAP, UDDI, and WSDL. Boston: Addison-Wesley, 2002. 332 p.

NOY, N. F.; McGUINESS, D. L. **Ontology development 101: a guide to creating your first ontology**. Disponível em: <http://protege.stanford.edu/ publications/>. Acesso em: 14 mai. 2004.

NWANA, H. S. Software agents: an overview. **Knowledge Engineering Review**, v. 3, n. 11, 1996. p. 1-40.

ORGANIZATION FOR THE ADVANCEMENT OF STRUCTURED INFORMATION STANDARDS. **HL7 Approves web services profile and ebXML as 24-month DSTUs for messaging standard**. Disponível em: <http://xml.coverpages.org/ni2004-05-06-a.html>. Acesso em: 05 dez. 2004.

OBJECT MANAGEMENT GROUP. **Interoperability solutions for healthcare**. Disponível em: <http://www.omg.org/attachments/pdf/*CORBAmed*_bro.pdf>. Acesso em: 05 dez. 2004.

OBJECT MANAGEMENT GROUP. **Introduction to OMG's specifications**. Disponível em: <http://www.omg.org/gettingstarted/specintro.htm>. Acesso em: 05 dez. 2004.

OELLERMANN JUNIOR, William L. **Architecting web services**. New York: Apress, 2001. 654 p.

OGBUJI, U. **The Past, present and future of web Services**: part 1. Sep. 2002. Disponível em: <http://www.webservices.org/index.php/article/articleprint/ 663/-1/24/>. Acesso em: 14 mai. 2004.

OGBUJI, U. **The past, present and future of web services**: part 2. Oct. 2002. Disponível em: <http://www.webservices.org/index.php/article/articleprint/ 679/-1/24/>. Acesso em: 14 mai. 2004.

PAHL, C.; CASEY, M. Ontology support for web service processes. In: EUROPEAN SOFTWARE ENGINEERING CONFERENCE HELD JOINT WITH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, 9. and 11., Sep. 2003. Helsinki. **Proceedings**… New York: ACM, 2003. p. 208-216.

PAN, J.; CRANEFIELD, S.; CARTER, D. A lightweight ontology repository. In: INTERNATIONAL JOINT CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, 2., July 2003. Melbourne. **Proceedings**… New York: ACM, 2003. p. 632-638.

PAPAZOGLOU, M. P.; Georgakopoulos, D. Service-oriented computing. **Communications of the ACM**, New York, v. 46, n. 10, Oct. 2003. p. 25-28.

PELTZ, C. Web services orchestration and choreography. **IEEE Computer**, Oct. 2003. p. 46-52.

PORTLAND PATTERN REPOSITORY. **Design patterns**. Disponível em: <http://c2.com/cgi-bin/wiki?DesignPatterns>. Acesso em: 14 mai. 2004.

PREE, W. Meta patterns: a means for capturing the essentials of reusable object-oriented design. **Lecture Notes in Computer Science**, v. 821, 1994.

PREE, W. et al. Active guidance of framework development. **Software Concepts and Tools**, v. 16, Springer-Verlag, 1995. p. 94-103.

PREE, W. **Essential framework design patterns**. 1997. Disponível em: <citeseer.nj.nec.com/article/pree97essential.html>. Acesso em: 14 mai. 2004.

PREE, W. Rearchitecturing legacy systems: concept & case study. In: WORKING IFIP CONFERENCE ON SOFTWARE ARCHITECTURE, 1., Feb. 1999. San Antonio. **Proceedings**… Deventer: Kluwer, 1999.

PREE, W. Framelets: small is beautiful. In: FAYAD, Mohamed E.; SCHMIDT, Douglas C.; JOHNSON, Ralph E. **Building application frameworks**: object-oriented foundations of framework design. New York: John Wiley & Sons, 1999. p. 411-414.

PREE, W.; Koskimies, K. Framelets: small and loosely coupled frameworks. In: **ACM COMPUTING SURVEYS**. New York: ACM, v. 32, n. 1es, 2000. Article n. 6.

RAMBHIA, Ajay M. **XML distributed systems design**. Indianapolis: Sams, 2002. 404 p.

RASKIN, V.; NIRENBURG, S. Ontology in information security: a useful theoretical foundation and methodological tool. In: NEW SECURITY PARADIGMS WORKSHOP, Sep. 2001. Cloudcroft. **Proceedings**… New York: ACM, 2001. p. 53-59.

RHEINHEIMER, Letícia R. **JLearningServices**: um framework para serviços síncronos em ambientes para EAD. São Leopoldo, 2002. Trabalho de Conclusão de Curso (Bacharelado em Informática) – Universidade do Vale do Rio dos Sinos, Centro de Ciências Exatas e Tecnológicas.

RICKEL, J.; JOHNSON, W. Integrating pedagogical capabilities in a virtual enviroment agent. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS, 1., Feb. 1997. **Proceedings**… New York: ACM, 1997. p. 30-38.

RIST, O. A matter of preference. **SD Times**, n. 56, June 2002. Disponível em: <http://www.sdtimes.com/cols/winwatch_056.htm>. Acesso em: 14 mai. 2004.

RUSSELL, Stuart J.; NORVING, Peter. **Artificial intelligence**: a modern approach. Upper Saddle River: Prentice-Hall, 1995. 932 p.

SANTOS, Cássia Trojahn dos. **Um Ambiente Virtual Inteligente e Adaptativo Baseado em Modelos de Usuário e Conteúdo**. São Leopoldo, 2004. 131 p. Dissertação (Mestrado) – Universidade do Vale do Rio dos Sinos, Centro de Ciências Exatas e Tecnológicas, Programa Interdisciplinar de Pós-Graduação em Computação Aplicada.

SHANKS, G.; TANSLEY, E., WEBER, R. Using ontology to validate conceptual models. **Communications of the ACM**, New York, v. 46, n. 10, Oct. 2003. p. 85-89.

SCHMIDT, D.; FAYAD, M.; JOHSNON, R. Software patterns. **Communications of the ACM**, New York, v. 10, n. 39, Oct. 1997.

SEELY, Scott. **SOAP**: **cross platform web service development using XML**. Upper Saddle River: Prentice-Hall, 2002. 391 p.

SELIGER, R. **Overview of HL7's CCOW standard**. Disponível em: <www.hl7.org/library/committees/sigvi/ccow_overview_2001.doc>. Acesso em: 05 dez. 2004.

SMITH, David C. KidSim: programming agents without a programming language. **Communications of the ACM**, New York, v. 7, n. 37, July 1994. p. 55-67.

SNELL, James; TIDWELL, Doug; KULCHENKO, Pavel. **Programming web services with SOAP**. Beijing: O'reilly, 2002. 244 p.

SUN MICROSYSTEMS. **Using web services effectively**. 2002. Disponível em: <http://java.sun.com/blueprints/webservices/using/webservbp.html>. Acesso em: 14 mai. 2004.

TERAI, K.; IZUMI, N.; YAMAGUCHI, T. Coordinating web services based on business models. In: INTERNATIONAL CONFERENCE ON ELECTRONIC COMMERCE, 5., Oct. 2003. Pittsburgh. **Proceedings**… New York: ACM, 2003. p. 473-478.

TOKORO, M. Agents: towards a society in which humans and computer cohabitate. In: EUROPEAN WORKSHOP ON MODELLING AUTONOMOUS AGENTS, 6., Aug. 1994. Odense. **Lecture notes in computer science**. London: Springer-Verlag, v. 1069, 1994. p. 1-10.

TSAI, T. et al. Ontology-mediated integration of intranet web services. **IEEE Computer**, Oct. 2003. p. 63-71.

TURNER, M.; BUDGEN, D.; BRERETON, P. Turning software into a service. **IEEE Computer**, Oct. 2003. p. 38-44.

USCHOLD, M.; GRÜNINGER, M. Ontologies: principles, methods and applications. **Knowledge Engineering Review**, v. 11, n. 2, June 1996.

VAN DEN HEUVEL, W.; MAAMAR, Z. Moving toward a framework to compose intelligent web services. **Communications of the ACM**, New York, v. 46, n. 10, Oct. 2003. p. 103-109.

WAGNER, Dirk N. Software agents take the internet as a shortcut to enter society: a survey of new actors to study for social theory. **First Monday**, v. 5, n. 7, July 2000. Disponível em: <http://firstmonday.org/issues/issue5_7/ wagner/index.html>. Acesso em: 14 mai. 2004.

WAGNER, G. The Agent-object-relationship metamodel: towards a unified view of state and behavior. **Information Systems**, v. 5, n. 28, 2003.

WEIPPL, E.; ESSMAYR, W. Personal trusted devices for web services: revisiting multilevel security. **Mobile Networks and Applications**, n. 8, 2003. p. 151-157.

WEISS, Gerhard (Ed.). **Multiagent systems**: a modern approach to distributed artificial intelligence. Cambridge: MIT Press, 1999. 619 p.

WILLIAMS, A.; Padmanabhan, A.; Blake, M. B. Local consensus ontologies for B2B-oriented service composition. In: INTERNATIONAL CONFERENCE ON AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS, 2., July 2003. Melbourne. **Proceedings**… New York: ACM, 2003. p. 647-654.

WIRSZ, N. Overview of IT-standards in healthcare. **Electromedica**, n. 1, 2000. Disponível em: <http://www.med.siemens.com/medroot/en/news/electro/issues/pdf/heft_1_00_e/05wirsze.pdf>. Acesso em: 05 dez. 2004.

WOOLDRIDGE, M.; JENNINGS, N. R. Intelligent agents: theory and practice. **Knowledge Engineering Review**, v. 2, n. 10, 1995. p. 115-152.

W3C WEB SERVICES ARCHITECTURE WORKING GROUP. **Web services architecture requirements**: W3C working draft. Aug. 2002. Disponível em: <http://www.w3.org/TR/ 2002/WD-wsa-reqs-20020819>. Acesso em: 14 mai. 2004.

YANG, J. Web service componentization. **Communications of the ACM**, New York, v. 46, n. 10, Oct. 2003. p. 35-40.

ZAMBONELLI, Franco; OMICINI, Andrea. Open directions in agent-oriented software engineering. **UPGRADE**, [s.l.], v. 5, n. 4, Aug. 2004. p. 11-14. Disponível em: <http://www.upgrade-cepis.org/issues/2004/4/upgrade-vol-V-4.html>. Acesso em: 05 dez. 2004.

ZÚÑIGA, G. L. Ontology: its transformation from philosofy to information systems. In: INTERNATIONAL CONFERENCE ON FORMAL ONTOLOGY IN INFORMATION SYSTEMS, 2. Oct. 2001. Ogunquit. **Proceedings**... New York: ACM, 2001. p. 187-197.

ANEXO A – PUBLICAÇÃO *OOPSLA* 2004

WSAgent: an Agent Based on Web Services to Promote Interoperability Between Heterogeneous Systems in the Health Domain

Letícia R. Rheinheimer Júnior M. Martins Sérgio Crespo C. S. Pinto
Interdisciplinar Postgraduation Program on Applied Computing – UNISINOS
P. O. Box 15.064 – 91.501-970 – São Leopoldo – RS – Brazil
+55 51 590 8161
0401901@exatas.unisinos.br 0500936@exatas.unisinos.br crespo@exatas.unisinos.br

ABSTRACT
This paper describes a Software Agent called WSAgent, which combines technologies such as Web Services, Frameworks and Design Patterns in the construction of a bind to grant interoperability, reuse and flexibility between heterogeneous environments in the health domain.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architectures: *domain-specific architectures.*

General Terms
Design.

Keywords
Web Services, Frameworks, Design Patterns, heterogeneous environments.

1. INTRODUCTION
Web Services technologies are based on the evolution of component architecture. Its concepts come from distributed computing, originated in the advent of computer networks. These technologies are establishing an emergent paradigm called Service Oriented Computing. In this component-based architecture, services form the basic building blocks for software construction [7].

Internet brought up the need of new development strategies. Also, old development strategies had to be adapted. This scenario led into an increasing need to satisfy requirements as quality, reuse and interoperability. Using Web Services contribute to satisfy these requirements [8].

Frameworks and Design Patterns can help to maximize the benefits of using Web Services, allowing us to develop flexible software [3, 6]. Techniques of diverse research areas (e. g., Artificial Intelligence) can be applied.

OOPSLA'04, Oct. 24–28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.

Web Services are being even more used in corporations. One of their main advantages is the possibility of integrating existing systems with platform independence, obtaining an agile and flexible solution. Application integration is a need experienced by several enterprises [5].

Health is a rich field where we can apply all mentioned technologies, because in this area we can find several legacy systems as well as several systems under development, and clinical data administration has a growing demand [1].

2. WHAT ARE WEB SERVICES?
As defined by the W3C Web Services Architecture Working Group, a Web Service is "a software application identified by an URI (Universal Resource Identifier), whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other Software Agents using XML-based messages exchanged via Internet-based protocols" [9].

Web Services combine the best aspects of component-based development. Like software components, Web Services represent a black box functionality that can be reused without worrying about what language and environment will be used [2].

Due to the benefits derived from the standard XML interface, Web Services enable dynamic service binding and an increased cross-language and cross-platform interoperability [4].

3. WSAGENT
The goal of this work is to apply Web Services technologies to develop Software Agents, which will be responsible for heterogeneous systems integration.

A situation frequently found in hospitals is the following: a new system is being developed and needs information that is persisted in an older system (called legacy). The hospital, in many cases, only owns executable files of the legacy system, and the company that provided support for that system may not operate anymore. So, how can we integrate these systems with no dependence on the legacy code? We propose to use Web Services in a way to bind calls to Agents in some points of the new system, so these will communicate with other Agents bound to the legacy system's database. All desired information is obtained directly from the databases, avoiding code to be rewritten or adapted.

WSAgent has a unique architecture that is constituted by three layers: the first uses Web Services in the role of Software Agents to carry out the communication between systems; the second represents data using Ontologies and contains the business rules which will be used to process all data exchanged between systems; the third is constituted by a Framelet (a little Framework) which will be responsible for data persistence, storing exchanged information correctly in several databases.

This architecture is called unique because the operation of Software Agents is guided by information defined as Ontologies. So, an Agent that works with the Patient-Bed process, for example, can be configured to work with the Stock process only by plugging in a new Ontology layer, which defines structure and rules for specific data manipulation. The persistence layer used by all agents will be the same. This Framelet makes it possible to customize the desired kind of database system. In case of need, the Framelet can be extended to work with a new kind of database.

The global solution to be used by an organization consists of several instances of the described architecture: several Software Agents customized with specific Ontologies and business rules, customized for a specific database system, and linked to specific points in applications which will activate these Agents when a critical operation is started. A critical operation needs to obtain the most updated data of a legacy system.

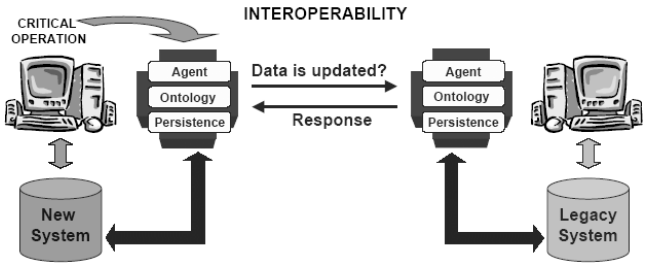


Figure 1. Agents reacting to a critical operation.

When a critical operation is executed (see Figure 1), a Web Service is invoked by the system and verifies the data of the system to which it is bound. When the verification is finished, another Web Service is invoked. It has to process the request, verify if there is new information in the database, prepare the data and send it to the calling Web Service. If no data is found, this notification is send to the calling Web Service. When the calling Web Service receives the response, it analyzes the information and, if it is needed, updates the database. When these steps are finished, control is returned to the application. This way of searching for updated information avoids overhead of network traffic. Agents monitoring systems would cause serious performance problems.

The main advantage of this solution is that it is fully distributed: Agents can be deployed in different application servers that, in turn, can be running in different computers.

At the present stage, a prototype is under development to validate the proposed solution. Java technologies and a spiral development methodology are being applied.

4. FUTURE WORK

After concluding the development of the prototype, tests for the Patient-Bed process will be done. These tests consist of applying the proposed solution in the integration of two systems that operate in the intranet of a hospital (a real case study).

When the desired results are obtained, the same work will be done for some other processes that involve the execution of critical operations. For these processes, only the development of a new Ontology layer will be necessary.

Based on these tests, the solution will be improved and validated.

5. ACKNOWLEDGMENTS

Thanks to the company *InfoSaúde Tecnologia de Informações Ltda.*[1], which develops solutions for several hospitals and contributes to this work with information and financial resources.

6. REFERENCES

[1] Bird, L. J. *Integrating Health Care Information using XML-Based Metadata.* Health Informatics Conference, South Australia, September 2002.

[2] Crespo, S. *Composition in WebFrameworks.* DSc. Thesis. Rio de Janeiro: PUC, 2000 (in portuguese).

[3] Fayad, M. E., and Schmidt, D. C., and Johnson, R. E. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. New York: John Wiley & Sons, 1999.

[4] Ferris, C., and Farrel, J. What Are Web Services? In *Communications of the ACM*, v. 46, n. 6, June 2003, p. 31.

[5] Foster, I., and Grossman, R. L. Data Integration in a Bandwidth-Rich World. In *Communications of the ACM*, November 2003, v. 46, n. 11, p.50-57.

[6] Gamma, E., et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading: Addison-Wesley, 1995.

[7] Papazoglou, M. P., and Georgakopoulos, D. Service-Oriented Computing. In *Communications of the ACM*, v. 46, n. 10, Ocotber 2003, p. 25-28.

[8] Turner, M., and Budgen, D., and Brereton, P. Turning Software into a Service. In *Computer*, October 2003, p. 38-44.

[9] W3C Web Services Architecture Working Group. *Web Services Architecture Requirements, W3C Working Draft.* August 19, 2002. Document available at http://www.w3.org/ TR/2002/WD-wsa-reqs-20020819 (last visited August 14, 2004).

[1] http://www.infosaude.com.br