UNIVERSIDADE DO VALE DO RIO DOS SINOS

Programa Interdisciplinar de Pós-Graduação em

Computação Aplicada

Andrêsa Vargas Larentis

Aruba: Uma Arquitetura para Geração de Serviços a partir de Sistemas Legados de Forma Não Intrusiva

UNISINOS

Andrêsa Vargas Larentis

Aruba: Uma Arquitetura para Geração de Serviços a partir de Sistemas Legados de forma Não Intrusiva

Dissertação 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

2007

L321a Larentis, Andrêsa Vargas

Aruba: uma arquitetura para geração de serviços a partir de sistemas legados de forma não intrusiva / por Andrêsa Vargas Larentis, 2007.

134 f. il. ; 30cm.

Dissertação (mestrado) -- Universidade do Vale do Rio dos Sinos, Programa de Pós-Graduação em Computação Aplicada, 2007.

"Orientação: Prof. Dr. Sérgio Crespo C. S. Pinto, Ciências Exatas e Tecnológicas".

1. *Web service*. 2. *Service oriented architecture* (SOA). 3. Arquitetura orientada a serviços. 4. Decomposição - Aplicação. 5. Integração - Aplicação. I. Título.

CDU 004.738.52:004.4

Catalogação na Publicação: Eliete Mari Doncato Brasil - CRB 10/1184

AGRADECIMENTOS

A Deus pela vida.

Ao meu marido, Sérgio Larentis, pelo apoio emocional e intelectual recebido no decorrer do mestrado.

Aos meus pais, Vilmar Vargas de Almeida e Sônia Maria de Almeida, e ao meu irmão, Adriano Vargas de Almeida, pelo apoio e motivação para que chegasse até aqui.

Ao meu orientador, Sérgio Crespo, pela orientação e conselhos construtivos que serviram para o desenvolvimento deste trabalho.

Ao Professor Dr., Jorge Luis Victória Barbosa, pela co-orientação que serviu para aprimorar o desenvolvimento do trabalho.

Aos meus amigos que souberam compreender a minha ausência.

RESUMO

A abordagem tradicional de construção de *softwares* baseia-se num modelo voltado para atender um conjunto de requisitos de negócios (normalmente um modelo de objetos ou componentes), porém, precisa estar preparado para atender necessidades de negócios que podem surgir no decorrer do processo. Assim, um modelo baseado em serviços pode representar uma nova forma de tecnologia para atender estas necessidades. No cenário brasileiro e internacional, o parque de sistemas de computação legado é muito expressivo. Estes sistemas se encontram em produção utilizando tecnologias muitas vezes já ultrapassadas que permitem uma pequena taxa de reuso. Portar estes sistemas para uma tecnologia baseada em serviços, visando obter reuso em grande escala, requer uma análise criteriosa do código existente e algumas customizações para disponibilizar serviços sem comprometer seu funcionamento.

Este trabalho propõe e implementa uma arquitetura denominada Aruba que permite a geração de serviços a partir de sistemas legados e o acesso a estes por meio de *Web Services* através de uma solução que não exija alteração no código existente, para posterior uso em uma SOA. A solução utiliza tecnologia de *Web Services*, as quais permitem uma descrição de serviços através de suas tecnologias padrões como WSDL e SOAP (todas baseadas em XML), e um banco de dados utilizado como repositório dos serviços gerados, capaz de armazenar e disponibilizar as informações destes serviços. Como forma de testar e avaliar a arquitetura foi realizado um estudo de caso, onde se procurou utilizar a arquitetura Aruba no desenvolvimento do processo de adequação do sistema legado para uma arquitetura baseada em serviço.

Palavras-chave: *Web Services*, SOA, Decomposição de Aplicações, Integração de Aplicações.

ABSTRACT

The traditional software development is based on a model focused in answering a set of business requirements (generally an object or component model), but it needs to be prepared to deliver solutions to business needs that may occur during the process. This way, a service based model may represent a new technology approach to meet these needs. At the Brazilian and international scenario there are many legacy systems. These systems are being currently in use and often are built on old technologies that do not allow much reuse. Porting these systems to a service based technology, aiming large scale reuse, needs analysis of the existing code and some customizations to make services available without compromising its functions.

This work explains an architecture called Aruba that allows service generation from legacy systems with no need of legacy code change. This solution uses the Web Services technology, which allows service description and use through WSDL and SOAP (all XML based), and uses a database as repository for the generated services. To test the architecture a case study was made where Aruba was used to translate a legacy system to a service oriented architecture environment.

***Key-Words**: Web Services, SOA, Application Decomposition, Application Integration*

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

API	*Application Program Interface*
ARUBA	*generAtion of seRvices in a non intrUsive way Based on legAcies*
BPEL	*Business Process Execution Language*
BPM	*Business Process Management*
CORBA	*Common Object Request Broker Arquitecture*
DCOM	*Distributed Component Object Model*
DDE	*Dynamic Data Exchange*
DOM	*Document Object Model*
EJB	*Enterprise Java Beans*
ERP	*Enterprise Resource Planning*
ESA	*Enterprise Services Architecture*
ESB	*Enterprise Service Bus*
FTP	*File Transfer Protocol*
HTML	*HiperText Markup Language*
HTTP	*HiperText Transfer Protocol*
IDC	*International Data Corporation*
IIOP	*Internet Inter-ORB Protocol*
IIS	*Web Microsoft Internet Information Services*
JMS	*Java Message Service*
JSP	*Java Server Pages*
MIME	*Multipurpose Internet Mail Extension*
OAR	*Options Analysis for Reengineering*
OASIS	*Organization for the Advancement of Structured Information Standards*
QoS	Qualidade de serviço
RMI	*Remote Method Invocation*
RPC	*Remote Procedure Call*
SEI	*Software Engineering Institute*
SGML	*Standard General Markup Language*
SMART	*Service-Oriented Migration and Reuse Technique*
SMIG	*Service Migration Interview Guide*

SMTP	*Simple Object Access Protocol*
SOA	*Service Oriented Architecture*
SOAP	*Simple Object Access Protocol*
TI	Tecnologia da Informação
UDDI	*Universal Distribution Discovery and Interoperability*
UML	*Unified Modeling Language*
W3C	*Word Wide Web Consortium*
WBS	*Work Breakdown Structure*
WCF	*Windows Communication Foundation*
WSDL	*Web Services Description Language*
WSE	*Web Services Enhancements*
XML	*Extensible Markup Language*
WCF	*Windows Communication Foundation*
WSDL	*Web Services Description Language*
WSE	*Web Services Enhancements*
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

A partir da segunda metade do século XX, as mudanças relacionadas ao desenvolvimento da Tecnologia da Informação (TI) ocuparam posição de destaque juntamente com seus impactos na esfera econômica. Com a evolução e o baixo custo dessas tecnologias ao longo desse período, a produção, o processamento e a disseminação de informações foram além de modificados, também aplicados com a criação de *softwares* que disponibilizam as informações necessárias para a comunicação empresarial [Bastos, 2007].

A abordagem tradicional de construção de *softwares* baseia-se num modelo voltado para atender um conjunto de requisitos de negócios (normalmente um modelo de objetos ou componentes), porém, precisa estar preparado para atender necessidades de negócios que podem surgir no decorrer do processo. A partir disso, a concepção de serviços representa uma nova forma de tecnologia que requer novos princípios técnicos de organização. Um serviço diferentemente de objetos e componentes, é concebido para satisfazer as regras de negócios de uma empresa, tornando-as mais ágeis. Sistemas conectados e orientados a serviços promovem um alinhamento estreito entre a solução de TI e as necessidades de negócio [Sehmi, 2006].

O mundo dos negócios demanda cada vez mais agilidade no desenvolvimento de artefatos de *softwares* e nem sempre a infra-estrutura existente pode atender à demanda. Assim, o conceito da *Service Oriented Architecture* (SOA) traz uma nova infra-estrutura para otimizar a construção de *software,* baseado num modelo de serviço, podendo ajudar na redução de custos garantindo agilidade nos processos de negócios [CIO, 2006].

No entanto, vários sistemas desenvolvidos nas empresas não são orientados a serviços. Assim, o objetivo a ser alcançado neste trabalho é transformar regras de negócios de sistemas já desenvolvidos, de forma a não alterar o código existente, em serviços a serem disponibilizados para uso de forma simples, com flexibilidade a mudanças e integração com outros serviços.

1.1 Motivação

Análises do *International Data Corporation* (IDC) [IDC, 2007] mostraram que 2007 foi o ano de grandes rupturas na indústria de TI. Diversos aspectos mudaram de forma radical o mercado tradicional, com alguns movimentos afetando a América Latina. Além disso, 2007 marcou o quarto ano de crescimento nos investimentos da América Latina em TI e, com um aumento de 6,7% nos investimentos em tecnologia no mundo, sendo que na América Latina este crescimento foi em torno de 12,8%.

As empresas possuem grande quantidade de sistemas legados que cobrem várias áreas de negócios, tais como: marketing, vendas, *help-desk*, estoque, faturamento, que foram desenvolvidos sob diferentes plataformas, linguagens de programação e arquiteturas. Qualquer alteração nestes sistemas pode muitas vezes custar caro e, também à migração para outra arquitetura requer que o sistema seja reconstruído na maioria das vezes. Não reaproveitar a lógica de negócio existente e no caso de uma alteração ou migração, propor que a lógica de negócio seja analisada como serviço, pode colaborar ainda mais para elevação dos custos que envolvem este desenvolvimento. Uma solução para promover o reaproveitamento da lógica de negócio é promover a sua decomposição e reuso.

Grandes fornecedores de *software* já estão disponibilizando no mercado suas plataformas que permitem a adoção de SOA nas empresas. A IBM disponibilizou a plataforma IBM *WebSphere*[1] como solução SOA. A Oracle disponibilizou o Oracle *SOA Suite*[2] que possibilita a adoção e gerenciamento de SOA. A Microsoft disponibilizou o *BizTalk Server* 2006[3] como solução SOA. Estas soluções possuem requisitos distintos de usabilidade e também possuem alto custo de propriedade.

Tendo em vista o cenário acima, nota-se que transformar um sistema legado em serviços que possam ser utilizados por outros aplicativos envolve várias etapas, incluindo desde a criação de um repositório que contenha os dados pertinentes ao uso dos sistemas legados até a disponibilização dos serviços criados. Assim, disponibilizar o acesso a sistemas

[1] IBM WebSphere – http://www.ibm.com

[2] Oracle SOA Suite – http://www.oracle.com

[3] BizTalk Server 2006 – http://www.microsoft.com

legados sem necessidade de codificação e reutilizar a lógica de negócio existente sem comprometer seu funcionamento motivam a criação da arquitetura Aruba (*generAtion of seRvices in a non intrUsive way Based on legAcies*) proposta neste trabalho.

1.2 Problema

Com os grandes avanços na área de TI, surge a necessidade da indústria de *software* fornecer ferramentas que atendam a demandas de negócios das empresas, construindo *softwares* com melhor desempenho, manutenibilidade e reusabilidade. Hoje, a maioria destas ferramentas possui um alto custo de propriedade e acabam inibindo as empresas neste investimento. Com o surgimento de novas tecnologias, o mercado de *software* ainda permanece em estágios iniciais em relação ao desenvolvimento de soluções que comportem estas novas tecnologias, e as empresas pouco se manifestam. Algumas questões são levantadas quanto a isso: Qual o custo para as empresas migrarem os sistemas legados que atendem suas áreas de negócio para SOA? Existe uma forma de não comprometer o funcionamento destes sistemas?

Ferramentas estão surgindo ao longo dos últimos anos. Todas possuem uma quantidade de soluções que possibilitam a migração para SOA, entretanto são consideradas intrusivas, pois fornecem soluções que precisam de alguma forma, alterar os sistemas legados e adaptar o código existente para promover a integração para SOA. A figura 1.1, exemplifica o termo intrusivo utilizado anteriormente. A partir do sistema legado, seu código fonte é reestruturado, de modo que um novo pacote de código seja gerado, através da decomposição de métodos. Após isso, os *Web Services* são criados baseando-se no código reestruturado.

Ao contrário da abordagem apresentada na figura 1.1, é importante destacar que o termo “Não intrusivo”, ao contrário do “Intrusivo”, significa não alterar o código fonte de um sistema.

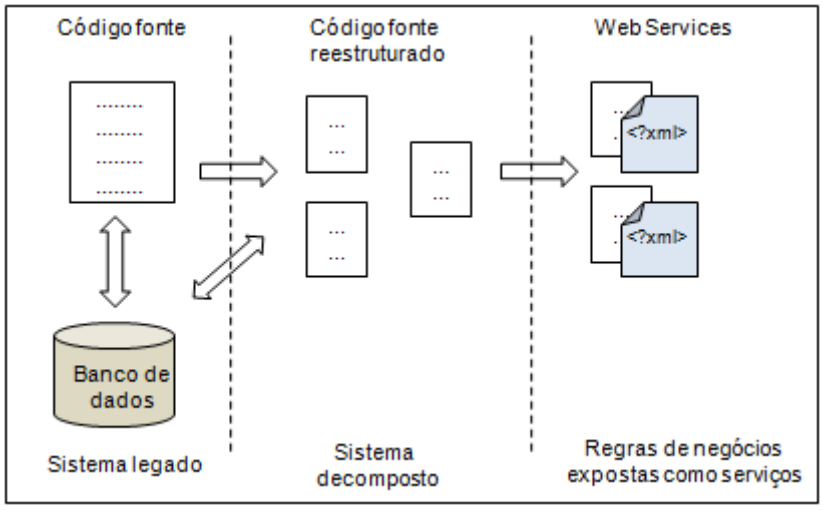

Figura 1.1 – Código fonte original alterado e *Web Services* intrusivo ao código.

O presente trabalho busca respostas para as seguintes perguntas:

"Como possibilitar que sistemas legados ou parte deles sejam decompostos em serviços para uso em SOA";

"Como disponibilizar o acesso a sistemas legados de forma não intrusiva, ou seja, que não exija alteração no sistema a ser acessado, com o objetivo de mapear funcionalidades de negócios em serviços sob a forma de *Web Services*".

Essas duas perguntas sintetizam os problemas que o presente trabalho busca auxiliar e amenizar.

1.3 Questão de Pesquisa

A questão central deste trabalho é: como permitir a geração de serviços, a partir de sistemas legados e, permitir o acesso a estes por meio de *Web Services,* através de uma solução não intrusiva, para posterior uso em uma SOA, viabilizando reuso de forma a preservar o investimento feito nos sistemas legados?

1.4 Objetivos do Trabalho

O objetivo deste trabalho é permitir que sistemas já existentes em uma corporação possam ter suas regras de negócios reutilizadas em uma SOA, sem que seja necessário alterar o código

fonte existente para geração de serviços, desde que sua arquitetura seja Java, .Net ou para suporte a *Dynamic Data Exchange* (DDE), ou seja, sistemas que permitem integração apenas através de uma interface. Para alcançar este objetivo, criou-se uma interface, com o objetivo de permitir que funcionalidades dos sistemas legados sejam mapeadas, assim como, uma arquitetura que será responsável pela tradução destas funcionalidades para *Web Services* com o intuito de disponibilizá-las sob o conceito de serviços.

A figura 1.2 exemplifica como a solução proposta neste trabalho gerou os *Web Services* sem necessitar reestruturar o código fonte original do sistema, de forma que estes fiquem armazenados num repositório que seja transparente para o sistema original.

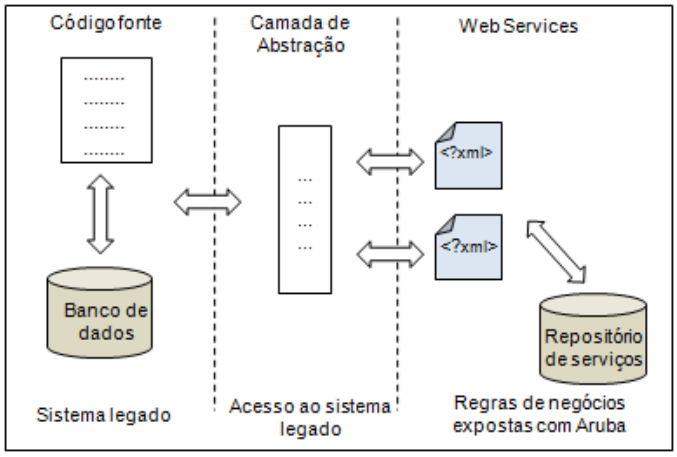

Figura 1.2 – Código fonte original mantido, *Web Services* não intrusivo.

Como avaliação do trabalho, criou-se um estudo de caso utilizando um protótipo da arquitetura, onde um sistema legado terá alguns métodos mapeados como serviços (*Web Services*). Por fim, estes serviços estarão disponíveis para consulta na interface da arquitetura demonstrando que, o mapeamento através de uma interface de configuração é simples e representa uma considerável economia no que diz respeito ao reuso destes serviços em SOA.

Além dos objetivos citados, o presente trabalho pretende alcançar os seguintes objetivos, mais específicos:

- Realizar um estudo aprofundado sobre as tecnologias necessárias para o desenvolvimento de *Web Services* e SOA;
- Estudar os conceitos de decomposição de serviços e integração de aplicações, apresentando as principais técnicas utilizadas;

- Possibilitar que sistemas legados definidos no escopo do trabalho possam ser tratados com o conceito de serviço;
- Propor e implementar uma arquitetura de geração de serviços que dê suporte a não intrusividade, gerando *Web Services* para estes serviços a partir do mapeamento feito por uma interface;
- Gerar um protótipo de interface que permita ao usuário mapear funcionalidades dos sistemas legados para *Web Services*;
- Avaliar o potencial da arquitetura desenvolvida em um estudo de caso, onde a mesma será avaliada através de sua utilização em uma aplicação.

1.5 Organização do Trabalho

Este trabalho está organizado em oito capítulos, sendo que no primeiro encontra-se a introdução. Os demais capítulos são descritos a seguir:

- Capítulo 2: Revisão Bibliográfica – descreve as tecnologias utilizadas no desenvolvimento do trabalho. O texto descreve a tecnologia de *Web Services*, SOA, Decomposição de Serviços e Integração de Aplicações;
- Capítulo 3: Trabalhos Relacionados – apresenta uma plataforma que fornece suporte a integração de sistemas legados da IBM, o IBM *WebSphere*, a plataforma da Microsoft, o *BizTalk Server* 2006, a plataforma da Oracle, o Oracle *SOA Suite*, e o SAP *NetWeaver*, todas as plataformas possuem objetivos semelhantes para integração de sistemas legados e que envolvem o uso das tecnologias descritas no capítulo 2. O capítulo também apresenta dois trabalhos acadêmicos relacionados com o que foi definido neste trabalho;
- Capítulo 4: Aruba – apresenta o modelo da arquitetura Aruba, seu projeto e os modelos das interações entre os componentes da arquitetura;
- Capítulo 5: Implementação – apresenta o projeto de implementação do protótipo da arquitetura para sua utilização;
- Capítulo 6: Estudo de Caso – apresenta um estudo de caso da utilização da arquitetura e os resultados obtidos;

- Capítulo 7: Conclusão – apresenta as conclusões e considerações finais, incluindo: comparações com os trabalhos relacionados, limitações e trabalhos futuros;
- Capítulo 8: Referências – lista as fontes de pesquisa utilizadas na realização deste trabalho.

2 REVISÃO BIBLIOGRÁFICA

Neste capítulo são apresentados os conceitos de *Web Services,* assim como, as principais tecnologias utilizadas na sua arquitetura, XML, WSDL, SOAP e UDDI. O capítulo apresenta a SOA, abordando os principais conceitos, componentes e funcionalidades. Por fim, são apresentados dois temas: Decomposição de Serviços e Integração de Aplicações, que estão relacionados à questão de pesquisa deste trabalho, de modo a familiarizar o leitor com estes conceitos auxiliando no entendimento deste trabalho.

2.1 *Web Services*

A necessidade de conectar informações e processos mudaram a forma como o *software* vem sendo desenvolvido. Sistemas bem-sucedidos de (TI) exigem cada vez mais interoperabilidade entre plataformas e serviços flexíveis que possam evoluir facilmente com o tempo. Segundo o *Word Wide Web Consortium*[4] (W3C), a tecnologia de *Web Services* fornece um mecanismo padrão de interoperabilidade entre diferentes aplicações de *softwares*, executando em uma variedade de plataformas e/ou *frameworks* [W3C, 2007].

Os *Web Services* são aplicações modulares que podem ser descritas, publicadas e invocadas sobre uma rede, geralmente a Web. Ou seja, é uma interface que descreve uma coleção de operações que são acessíveis pela rede através de mensagens em formato XML padronizadas. Sua estrutura arquitetural permite a comunicação entre aplicações, assim, um serviço pode ser invocado remotamente, ou ser utilizado para compor um novo serviço juntamente com outros [Hansen, 2003].

Um *Web Service* pode ser visto como um componente de *software* independente de implementação ou plataforma, que pode ser: descrito utilizando-se uma linguagem de descrição de serviços; publicado em um registro de serviço; descoberto através de um mecanismo de busca padrão; ser invocado através de uma *Application Program Interface* (API), via rede; e ser combinado com outros serviços [Newcomer, 2002].

---

[4] O site oficial do W3C pode ser acessado em: <http://www.w3.org/>

Segundo definição do W3C, um *Web Service* é: "uma aplicação de *software* identificada por um URI[5], 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, 2007].

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 [Crespo, 2000], [Graham et al., 2002], [Hansen, 2003]. Os *Web Services* permitem uma integração de serviços de maneira mais rápida e eficiente [Kreger, 2001].

Um *Web Service* descreve funcionalidades específicas do "negócio" com o propósito de fornecer um caminho para a utilização deste serviço. A exposição de um serviço se faz por meio de: identificar ou definir funções de valor no negócio ou processos; definir uma interface baseada no serviço para os processos; e, descrever estas interfaces em um formato baseado na Web. Para tornar um serviço disponível na Web normalmente é necessário: publicar a interface do serviço para que este possa ser encontrado e utilizado; aceitar requisições e enviar respostas usando protocolo padrão e mensagens em formato XML; e, fazer uma ligação entre requisições externas e implementações das funções dos negócios [Hansen, 2003].

Web Services são acessados por outras aplicações via protocolos de transporte como: *Hiper Text Transfer Protocol* (HTTP), *File Transfer Protocol* (FTP) e *Simple Mail Transfer Protocol* (SMTP) e não via protocolos específicos de modelos de objetos: *Distributed Component Object Model* (DCOM), *Remote Method Invocation* (RMI) ou *Internet Inter-ORB Protocol* (IIOP) [Oellermann, 2001].

---

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

2.1.1 Arquitetura de *Web Services*

A arquitetura de *Web Services* para disponibilização e acesso aos serviços está baseada nas interações de três papéis: provedor, solicitante e registro de serviço [Kreger, 2001], [Oellermann, 2001]. A figura 2.1 apresenta as interações entre esses papéis.

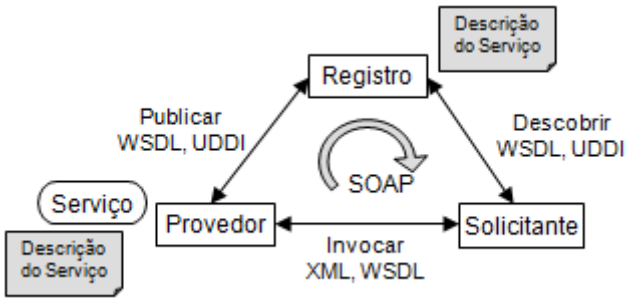

Figura 2.1 – Papéis, operações e artefatos de *Web Services* [Kreger, 2001].

O provedor de serviço é a plataforma acessada na solicitação do serviço. Trata-se da entidade que cria o *Web Service*, sendo responsável por fazer sua descrição em 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. Com a descrição de serviço é possível descobrir onde está um *Web Service* e como invocá-lo. O funcionamento inicia quando 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 criada, apresentando como enviar cada mensagem para o endereço onde o *Web Service* está localizado.

O solicitante de serviço é uma aplicação que invoca ou inicia uma interação com um serviço. Pode ser um *browser* ou um programa sem interface com o usuário, por exemplo, outro *Web Service*. Um solicitante de serviço encontra uma descrição de serviço, ou consulta o registro de serviço para o tipo 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 ocorre via rede. As tecnologias e padrões utilizados no desenvolvimento de um *Web Service* podem ser representados conforme a figura

2.2, traduzindo esse cenário para um conjunto de camadas conceituais descritas a seguir [Kreger, 2001], [Rheinheimer, 2004]:

- **Camada de rede**: camada base que abrange os protocolos de transporte como: HTTP, FTP e SMTP, podendo ser utilizada para implementação de necessidades das aplicações, tais como: disponibilidade, desempenho, segurança e confiabilidade;
- **Mensagem**: as mensagens têm como base a tecnologia XML e o protocolo SOAP para realizar a troca de mensagens entre provedor, solicitante e registro de serviço;
- **Descrição do serviço**: a descrição do serviço é feita com uma linguagem específica denominada WSDL, que define uma interface e mecanismos de interação dos serviços. Também define descrições adicionais como contexto, qualidade do serviço e o relacionamento de serviço para serviço;
- **Publicação e descoberta do serviço**: essas camadas utilizam o registro UDDI para fazer a descoberta e a publicação de informações sobre *Web Services*.

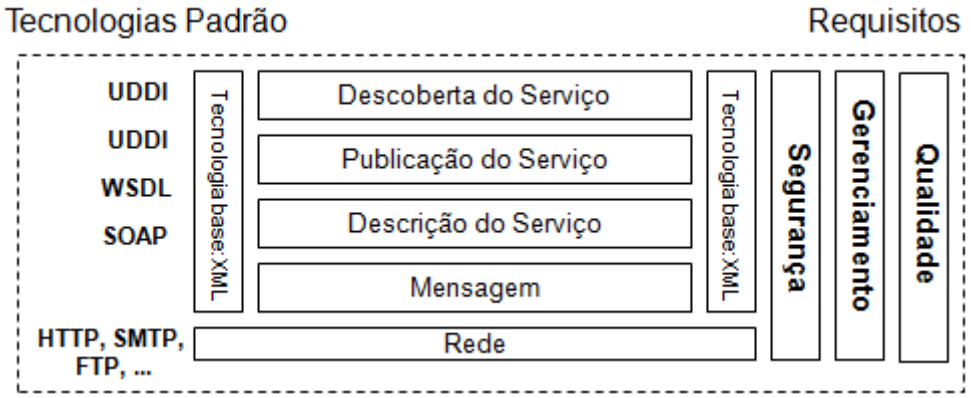

Figura 2.2 – Camadas conceituais de *Web Services* [Kreger, 2001].

2.1.2 Tecnologias padrão utilizadas na arquitetura de *Web Services*

A seguir, serão descritas as tecnologias padrão utilizadas na construção de *Web Services*: WSDL, SOAP e UDDI. Essas tecnologias são baseadas em XML e permitem invocar um serviço sem a necessidade de conhecer a plataforma ou linguagem de programação usada na sua construção.

2.1.2.1 XML

Extensible Markup Language (XML) é uma das tecnologias-chave para a construção e utilização de *Web Services* [Roy and Ramanujan, 2001]. XML foi criada em 1996 pelo W3C, liderado por Jon Bosak, da Sun Microsystems, sendo um subconjunto da *Standard General Markup Language* (SGML). SGML é um padrão complexo para descrever a estrutura do conteúdo de documentos [Sampaio, 2006].

XML é uma linguagem para organização dos dados de um documento, em formato textual, que contém marcadores como um arquivo *Hypertext Markup Language* (HTML). Esses marcadores são definidos através de uma linguagem onde sua sintaxe é baseada em marcas (tags). As linguagens de marca são bem simples e de fácil entendimento. Para uso das tags não existe um padrão definido, assim, qualquer pessoa pode definir conforme a necessidade de uso [Sampaio, 2006]. A figura 2.3 apresenta um exemplo da estrutura básica de um documento XML.

```
<?xml version="1.0"?>
<curso tipo="programacao">
      <nome>Introducao do XML</nome>
      <descricao>Introducao a linguagem XML</descricao>
      <carga>60 horas</carga>
</curso>
```

Figura 2.3 – Estrutura básica de um documento XML.

No exemplo da figura 2.3, se o texto for editado em um bloco de notas e salvo com extensão .XML, o arquivo poderá ser aberto e visualizado em um *browser*, por exemplo, *Internet Explorer*. Os arquivos XML são aceitos por programas capazes de processar XML.

2.1.2.2 WSDL

A *Web Service Description Language*[6] (WSDL) é uma linguagem em formato XML para descrição de interface dos serviços, de forma que outros programas possam interagir com esses serviços. WSDL descreve um serviço como uma coleção de operações que podem ser acessadas através de mensagens. Para sua utilização com *Web Services* deve existir um arquivo WSDL, escrito em XML, para cada *Web Service*, cuja função é descrever as operações que o *Web Service* realiza [Hansen, 2003], [Newcomer, 2002].

A WSDL está dividida em três elementos principais: definições de tipo de dados, operações abstratas e protocolos de ligação. Cada um desses elementos podem ser especificados em documentos XML diferentes e importados em diferentes combinações, para criar a descrição final de um *Web Service,* ou definidos juntos em um único arquivo XML. A definição do tipo de dados determina a estrutura e o conteúdo das mensagens. As operações abstratas determinam as operações possíveis e o protocolo de ligação determina a forma de transmissão das mensagens pela rede até os destinatários [Newcomer, 2002].

A descrição de um serviço consiste de duas partes: definição da implementação do serviço e definição da interface do serviço [Hansen, 2003]. A figura 2.4 apresenta essas definições.

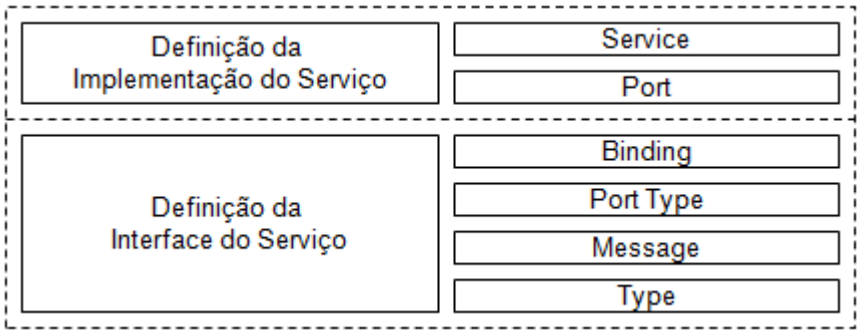

Figura 2.4 – Camada de descrição dos serviços [Hansen, 2003].

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

A camada de definição da interface do serviço contém a definição de serviço WSDL, permitindo que uma interface possa ser utilizada, instanciada e referenciada por múltiplas definições de implementação de serviços, incluindo diretivas:

- *wsdl:binding* - descreve protocolos, formato de dados, 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 camada de definição de implementação do serviço descreve como uma interface de serviço é implementada por um provedor, ou seja, 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*. Um arquivo de implementação descreve onde o *Web Service* está instalado e como é acessado e, além disso, a WSDL especifica extensões para ligações com protocolos e formatos de mensagem como SOAP, HTTP GET/POST e *Multipurpose Internet Mail Extensions* (MIME).

O trecho a seguir define um exemplo simplificado de uma descrição WSDL.

```
<message name="getMethodSearchRequest">
     <part name="name" type="xsd:string"/>
</message>
<message name="getMethodSearchResponse">
     <part name="value" type="xsd:string"/>
</message>
<portType name="methodSearchs">
     <operation name="getMethodSearch">
          <input message="tns:getMethodSearchRequest"/>
          <output message="tns:getMethodSearchResponse"/>
     </operation>
</portType>
```

Figura 2.5 – Exemplo de documento WSDL.

Nos elementos *message* estão definidas as partes de uma mensagem e os tipos de dados associados, ou seja, os elementos de dados de uma operação. No exemplo, a definição de uma mensagem nomeada *getMethodSearchRequest* possui um elemento chamado *name*, do

tipo string. Outra mensagem nomeada *getMethodSearchResponse*, possui um elemento chamado *value*, do tipo string. Comparando com a programação tradicional, é o mesmo que ter uma função *getMethodSearchRequest* com o parâmetro *name* e outra função *getMethodSearchResponse* 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, *methodSearchs* é definido como um elemento de operação do *Web Service*, onde *getMethodSearch* é o nome da operação. A operação *getMethodSearch* tem uma mensagem de entrada chamada *getMethodSearchRequest* e uma mensagem de saída chamada *getMethodSearchResponse*. Fazendo uma comparação com a programação tradicional, *methodSearchs* seria uma biblioteca de funções, e a operação *getMethodSearch* seria uma função com *getMethodSearchRequest* como parâmetro de entrada e *getMethodSearchResponse* como parâmetro de retorno.

2.1.2.3 SOAP

O protocolo *Simple Object Access Protocol*[7] (SOAP) permite a comunicação entre diversas aplicações em um ambiente distribuído e descentralizado. A comunicação é realizada através de trocas de mensagens, transmitidas em formato XML, incluindo parâmetros usados na chamada, bem como os dados de resultados. Isto significa que as mensagens podem ser utilizadas e compreendidas por quase todas as plataformas de *hardware*, sistemas operacionais, linguagens de programação e equipamentos de rede [Hansen, 2003].

Um pacote SOAP consiste de quatro partes [Newcomer, 2002], [Hansen, 2003]:

- **Envelope**: guarda o conteúdo da mensagem, quem poderá tratá-la e se o tratamento é opcional ou obrigatório. A estrutura de mensagem SOAP encapsula os elementos sintáticos da mensagem. O envelope contém os seguintes subelementos: cabeçalho – contém os atributos opcionais das mensagens; corpo - contém os dados da mensagem em XML;

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

- **Codificação**: responsável por definir mecanismos de serialização que podem ser utilizados para trocar instâncias ou tipos de dados definidos por uma aplicação;
- ***Remote Procedure Call***[8] (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.

A figura 2.6 apresenta as três partes principais das mensagens SOAP: envelope, cabeçalho e corpo. O envelope é obrigatório e marca o início e o fim das mensagens. O cabeçalho é opcional e pode conter um ou mais blocos com atributos da mensagem. O corpo também é obrigatório e contém um ou mais blocos contendo a mensagem propriamente dita [Newcomer, 2002].

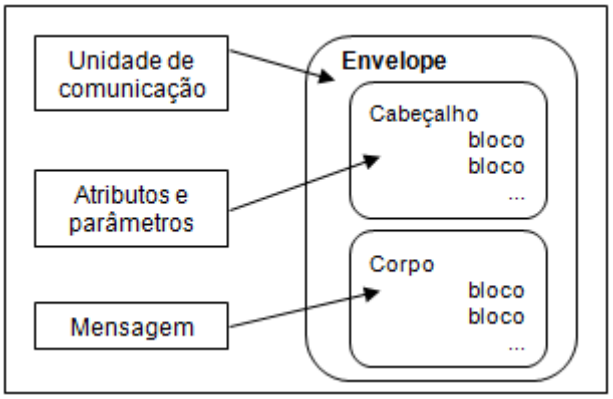

Figura 2.6 – Estrutura do envelope SOAP [Newcomer, 2002].

Quando um *Web Service* é implementado, os dados contidos nas mensagens SOAP devem ser interpretados, já que o SOAP não define o serviço, apenas contém dados suficientes para que o processador SOAP possa reconhecê-lo. O SOAP é definido com um nível de abstração suficiente para abranger tanto documentos como interações RPC [Newcomer, 2002].

A invocação do serviço ocorre conforme apresentado na figura 2.7.

[8] Informações podem ser encontradas em: <http://www.faqs.org/rfcs/rfc1050.html>

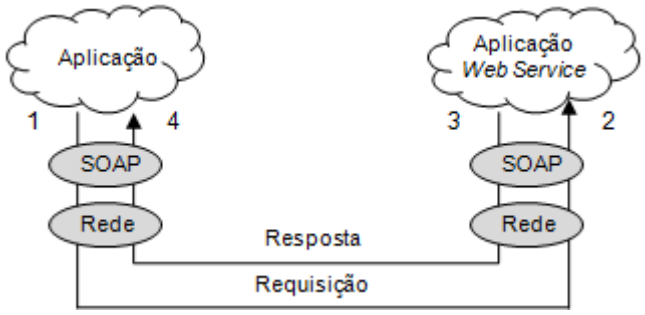

Figura 2.7 – Invocação do serviço utilizando SOAP [Kreger, 2001].

A aplicação (1) requisita uma mensagem SOAP e invoca a operação do serviço através de um provedor de *Web Services*. O solicitante de serviço apresenta a mensagem junto com o endereço de rede do provedor de *Web Service*. A infra-estrutura de rede (2) entrega a mensagem para um servidor SOAP. O servidor SOAP redireciona a mensagem requisitada para o provedor de serviço *Web Service*. O servidor Web (3) é responsável por processar uma mensagem de requisição e formular a resposta. Quando a mensagem XML chega ao nodo requisitante, é convertida para uma linguagem de programação, sendo entregue para a aplicação (4) [Kreger, 2001].

2.1.2.4 UDDI

A especificação *Universal Description, Discovery and Integration*[9] (UDDI) tem como objetivo criar um padrão para a descoberta de serviços. Com UDDI é possível localizar um serviço, constituindo de uma especificação técnica para descrever, descobrir e integrar *Web Services* [Newcomer, 2002].

O UDDI é constituído de duas partes: o UDDI *Project* que é uma especificação técnica utilizada para construir e distribuir *Web Services*, a qual permite que as informações sejam armazenadas em um formato XML especifico; e o UDDI *Business Registry*, que é uma implementação operacional completa da especificação UDDI [Hansen, 2003], [Rheinheimer, 2004].

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

O UDDI *Project* é considerado o componente central que manipula o registro global e público UDDI *Business Registry*. Toda informação mantida no UDDI *Business Registry* está disponível para consultas em geral.

A informação oferecida pelo UDDI *Business Registry* consiste de três componentes [Rheinheimer, 2004]:

- *white pages*: são incluídas informações gerais sobre a empresa específica, tais como: endereço, contato e identificadores conhecidos;
- *yellow pages*: são incluídos dados de classificação gerais da empresa ou serviço oferecido, tais como: categorização industrial;
- *green pages*: são incluídas informações técnicas sobre *Web Services*.

A implementação UDDI é um servidor de registro que fornece um mecanismo para publicar e descobrir *Web Services*. O registro UDDI contém informações categorizadas sobre empresas e serviços que elas oferecem, e associações destes serviços com especificações dos *Web Services* podem ser feitas em WSDL através do próprio registro [Hansen, 2003], [Rheinheimer, 2004].

O acesso público do registro pode ser efetuado via Internet e o acesso privado pode ser efetuado em Intranets de empresas, por exemplo. Um registro UDDI pode também ser acessado por aplicações, via código ou através de alguma interface.

O modelo de informação principal utilizado pelo registro UDDI é definido através de *XML Schema*, definindo quatro tipos de informações: negócio, serviço, ligação e a especificação do serviço [Hansen, 2003], [Rheinheimer, 2004].

A informação referente ao registro de um serviço consiste de cinco tipos de estruturas de dados [Newcomer, 2002], [Hansen, 2003], [Rheinheimer, 2004]. Esta divisão por tipos de informação fornece partições simples para auxiliar na rápida localização e compreensão das diferentes informações que compõem o registro. A figura 2.8 apresenta as cinco estruturas.

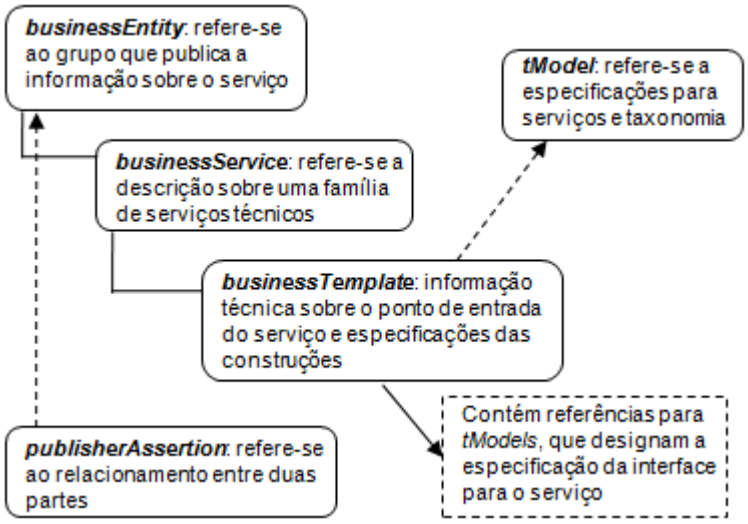

Figura 2.8 – Modelo de estrutura UDDI [Hansen, 2003].

O *businessEntity* fornece informação sobre uma empresa tais como: nome, descrição e endereço, podendo conter um ou mais *businessServices, r*epresentando toda a informação conhecida sobre uma empresa específica ou informações descritivas sobre uma entidade, bem como os serviços que ela fornece.

O *businessService* define descrições técnicas e de negócios tais como: nome e descrição para um ou grupo de *Web Services* relacionados, representando uma classificação lógica de serviço. Cada estrutura *businessService* pertence a uma única estrutura *businessEntity*.

O *bindingTemplate* define descrições técnicas de como e onde acessar um *Web Service* específico, fornecendo suporte para que se possam acessar os serviços remotamente, e definem o suporte a tecnologias, parâmetros específicos da aplicação e os arquivos de configuração.

A estrutura *tModel* é representada por meio de metadados (dados sobre dados). Seu propósito é fornecer um sistema de referência para os documentos WSDL.

O *publisherAssertion* permite que se possa associar estruturas *businessEntity*, de forma a obter uma melhor identificação. Ambas as estruturas devem publicar exatamente a mesma informação, para que o relacionamento torne-se visível.

Cabe salientar que 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 fornece suporte a criação de descrições WSDL de forma direta.

Uma descrição WSDL completa consiste da: combinação dos documentos de interface, que pode ser publicada no registro usando um *tModel*, o que deve ser feito antes da implementação ser publicada como *businessService;* e, da implementação do serviço, que é publicada no registro UDDI como um *businessService* ou dentro de um *bindingTemplate* (ou mais de um) [Hansen, 2003], [Rheinheimer, 2004].

2.2 *Service Oriented Architecture* (SOA)

Um caminho para constituir um sistema é considerá-lo uma composição de um conjunto de interações de serviços, assim, cada serviço fornece acesso a um conjunto de funcionalidades bem-definidas. O sistema como um todo é projetado e implementado, e interações entre os serviços são mantidas. Expor uma funcionalidade como um serviço é a chave para a flexibilidade [Brown et al., 2002].

Como os sistemas de informações têm crescido exponencialmente, as empresas têm construído arquiteturas de *softwares* cada vez mais complexas, onde precisam responder rapidamente as novas exigências de negócios, reduzirem continuamente os custos de TI, e ao mesmo tempo absorver e integrar novos negócios de parceiros e clientes. Arquiteturas tradicionais alcançaram o limite de suas capacidades e assim, uma nova arquitetura se faz necessária para maximizar o reuso e flexibilidade nos negócios [IBM, 2005].

SOA relaciona serviços e seus consumidores, que representam um processo de negócio. Os serviços podem ser acessados pelo nome via uma interface, e os consumidores acessam os serviços disponíveis via interface de serviço, por exemplo, *Web Services*. Essa nova arquitetura está substituindo aos poucos as arquiteturas monolíticas, por exemplo, princípios de projeto, para novas aplicações de negócios [Gartner, 2003].

As definições apresentadas na próxima seção apresentam abordagens de diferentes autores sobre a SOA.

2.2.1 Definindo SOA

Algumas definições de SOA são encontradas na literatura:

- "SOA é um novo paradigma de desenvolvimento de aplicações cujo objetivo é criar módulos funcionais chamados de serviços, com baixo acoplamento e permitindo a reutilização de código" [Sampaio, 2006].
- "SOA é uma forma de tecnologia arquitetural que adere aos princípios da orientação a serviços. Quando realizada através da plataforma de tecnologia de *Web Services*, SOA estabelece um potencial para fornecer suporte e promover estes princípios durante todo o processo de negócios e domínio de automação de uma empresa" [Erl, 2006].
- O W3C define SOA como: "uma forma de arquitetura de sistemas distribuídos" [W3C, 2007]. Estes sistemas, por sua vez, são caracterizados pelas seguintes propriedades:
    - Visão lógica: o serviço é uma visão abstrata e lógica dos programas reais, banco de dados, processos de negócio, etc., definidos em termos do que faz, realizando tipicamente uma operação em nível de negócios;
    - Orientação por mensagens: o serviço é formalmente definido em termos de trocas de mensagens entre provedor e solicitante, e não nas propriedades deles, já que são abstratas na implementação de SOA;
    - Orientação por descrição: o serviço é descrito por um metadado. A descrição suporta a natureza pública de um SOA: somente aqueles detalhes que são expostos publicamente e importantes para o uso do serviço devem ser incluídos na descrição; A semântica de um serviço deve ser documentada, diretamente ou indiretamente, por sua descrição;
    - Granularidade: serviços tendem a usar um pequeno número de operações como mensagens relativamente grandes e complexas;
    - Orientação por rede: serviços tendem a ser orientados para uso em rede, embora isto não seja um requisito absoluto;
    - Plataforma neutra: mensagens são enviadas em uma plataforma neutra, em formato padronizado, por exemplo, XML, e entregue através de interfaces.

A figura 2.9 apresenta um modelo de arquitetura do W3C que encapsula diferentes conceitos, como: política, mensagens, recursos e ações de forma a traduzir um modelo geral de SOA [W3C, 2007].

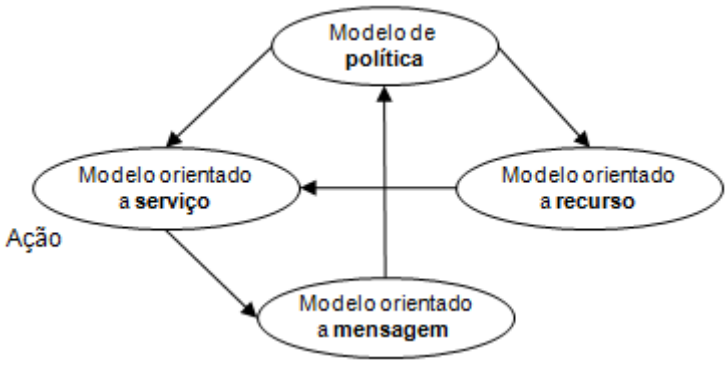

Figura 2.9 – Modelo W3C SOA [W3C, 2007].

Os quatros modelos da arquitetura são:

- **Modelo orientado a mensagem**: define a mensagem em termos de conteúdo (cabeçalho e corpo), transporte de entrega, solicitante e provedor do serviço;
- **Modelo orientado a recurso**: define recursos em termos de endereço (URI), representação e dono do recurso;
- **Modelo de políticas**: define a política em termos de recursos, aplicada também para descrições dos serviços;
- **Modelo orientado a serviço**: mais complexo de todos. Um serviço é oferecido e utilizado, sendo mediado por meio de trocas de mensagens.

2.2.2 Tecnologias que fundamentam SOA

Esta seção apresenta uma abordagem de como tecnologias como XML e *Web Services* estão sendo usadas para embasar a SOA e fundamentá-la de tal forma que possa ser aplicada.

2.2.2.1 XML e SOA

Através do uso de XML, torna-se possível unir significado e contexto para alguma parte da informação transmitida por meio dos protocolos da Internet [Newcomer, 2002].

A arquitetura de representação dos dados em XML estabelece uma camada principal para construção de uma SOA. Nesta camada, o XML estabelece o formato e a estrutura das mensagens que navegam por todos os serviços. Qualquer movimentação dentro de SOA não pode ser feita sem envolver XML [Rogers and Hendrick, 2005].

2.2.2.2 *Web Services* e SOA

O conceito de criar um *framework* de comunicação padrão, baseado na Web com tecnologia distribuída, para servir de ponte a enorme disparidade que existe entre e dentro das organizações, recebeu o nome de *Web Services*. Para dar suporte ao uso de *Web Services* surgiram os conceitos de WSDL, SOAP e UDDI, que completaram a primeira geração da família padrão para *Web Services* [Newcomer, 2002].

Web Services foi a base da criação de uma plataforma arquitetural separada, que alavancou um conjunto de benefícios desta tecnologia para realizar o conceito de serviços nas empresas. Assim, SOA ganhou popularidade com o conceito de serviços.

Componentes SOA podem ser criados usando qualquer tecnologia. Um modelo, inspirado pelo conjunto de padrões de *Web Services*, definiu SOA como uma arquitetura modelada em torno de três componentes básicos: provedor, solicitante e registro de serviço. Este modelo serviu apenas para ilustrar o conceito inicialmente formado para SOA, de uma perspectiva de arquitetura física, a variação de um SOA baseado em *Web Services* vai além desta primeira definição [Keen et al., 2004].

Com a evolução de XML e *Web Services*, SOA evoluiu também. Isto é resultado da relação entre numerosas iniciativas dirigidas por uma variedade de organizações de padrões e desenvolvimento de *software.* Os padrões de *Web Services* continuam a serem adotados em grande número e com isso fornecem suporte ao uso em SOA [Rogers and Hendrick, 2005].

2.2.3 Fundamentos de SOA

O termo “orientado a serviços” existe há algum tempo e tem sido usado em diferentes contextos e para diferentes propósitos, por exemplo, separação de interesses, que significa

decompor um sistema em um conjunto de partes menores, onde cada uma destas partes atenda um interesse ou uma parte específica de um problema [Sehmi, 2006].

Quando acoplado com "arquitetura", orientado a serviços leva uma conotação técnica. "SOA" é um termo que representa um modelo para organização e utilização de lógica de negócios distribuída que estão sob o controle de diferentes domínios proprietários. Os termos visibilidade, interação e efeito são chaves na SOA. A visibilidade introduz a possibilidade de compartilhar necessidades, a interação é a atividade que usa a lógica de negócios, e o efeito é o que o uso da lógica de negócios promove no mundo real. O mecanismo pelo qual as necessidades e a lógica de negócios são colocadas juntas é chamado de serviço [OASIS, 2006].

Sampaio define um serviço como sendo um "componente que atende a uma função de negócio específica para os clientes. O serviço recebe requisições e as respondem ocultando todo o detalhamento do seu processamento" [Sampaio, 2006].

Um serviço pode executar unidades completas de trabalho, não dependendo do estado de outros componentes externos, aumentando assim a sua reutilização, ou seja, um serviço executa uma função atômica (ou transação). Todas as etapas intermediárias devem ser gerenciadas apenas pelo serviço e não pelo solicitante. Serviços podem ser criados usando qualquer linguagem, tecnologia ou plataforma. Os itens a seguir apresentam exemplos de serviços [Sampaio, 2006]:

- Verificar a disponibilidade de vôos para uma determinada cidade;
- Efetuar a venda de um determinado produto;
- Reservar hotel para um cliente.

2.2.3.1 Princípios comuns da orientação a serviços

Um projeto de *Web Service* para SOA difere de outros *Web Services* criados para uso em outros ambientes de aplicações distribuídas porque seguem um conjunto de convenções. Existe um conjunto comum de princípios associados com orientação a serviços que estabelecem uma abordagem única de projeto para construir *Web Services* para SOA. Quando aplicados, estes princípios levam a uma padronização de *Web Services,* ao mesmo tempo que

preservam o baixo acoplamento das relações entre eles. Erl [Erl 2006] apresenta uma lista destes princípios:

- **Serviços são reusáveis**: serviços são projetados para suportar potencial reuso;
- **Serviços compartilham um contrato formal**: para os serviços interagirem, não necessitam compartilhar algo, mas um contrato formal que descreve cada serviço e define os termos das informações trocadas, como operações e mensagens;
- **Serviços são fracamente acoplados**: serviços devem ser projetados para interagir de forma independente e resistente a mudanças;
- **Serviços devem abstrair a lógica de negócios**: somente o que está exposto via contrato de serviço é que está visível para o solicitante do serviço, a lógica de negócio é invisível e irrelevante;
- **Serviços são passíveis de composição**: serviços podem compor outros serviços. Esta lógica pode ser representada por diferentes níveis de granularidade e promover reusabilidade e criação de camadas de abstração;
- **Serviços são autônomos**: a lógica governada por um serviço reside dentro de um bloco explícito. O serviço tem controle dentro deste bloco e não está dependente de outros serviços para executar esta governança;
- **Serviços são *stateless***: serviços não devem ser utilizados para gerenciar o estado das informações;
- **Serviços são passíveis de descoberta**: serviços levam suas descrições para serem descobertos e entendidos por solicitantes de serviços que podem estar habilitados para fazer uso da sua lógica de negócios.

A tecnologia de *Web Services* dá suporte a alguns destes princípios descritos anteriormente, porém, foram identificados quatro princípios que não são providos por esta tecnologia, que são: serviços são reusáveis; serviços são autônomos; serviços são *stateless*; serviços são passíveis de descoberta. A tabela 2.1 apresenta uma descrição que compara cada um destes princípios com a tecnologia de *Web Services* [Keen et al., 2004], [Erl, 2006].

Tabela 2.1 – Princípios da orientação a serviços suportados por *Web Services* [Erl, 2006].

Princípios da orientação a serviços	**Suporte *Web Services***
Serviços reusáveis	*Web Services* não são automaticamente reusáveis.
Serviços compartilham contrato formal	*Web Services* requerem o uso de descrições de serviços, tornando os contratos de serviço parte da sua comunicação.
Serviços são fracamente acoplados	*Web Services* são naturalmente fracamente acoplados através do uso das descrições dos serviços.
Serviços devem abstrair lógica de negócios	*Web Services* automaticamente emulam o modelo de *black box* dentro do *framework* de comunicação de *Web Services*, escondendo os detalhes da lógica de negócios.
Serviços são passíveis de composição	*Web Services* são naturalmente compostos.
Serviços são autônomos	*Web Services* não são autônomos. Isso requer muito esforço no projeto.
Serviços são *stateless*	*Stateless* é a condição preferida para *Web Services*.
Serviços são passíveis de descoberta	Não é suportado por *Web Services*. Deve ser implementada por uma arquitetura e pode ser considerada uma extensão para infra-estrutura de TI.

Abstração, possibilidade de composição, baixo acoplamento e a necessidade por contratos de serviços são características nativas de *Web Services* que estão em total alinhamento com os princípios correspondentes da orientação a serviços. Reusabilidade, autonomia, *stateless* e descoberta não são automaticamente providos por *Web Services*. Realizar estas tarefas requer esforço em relação ao modelo e ao projeto.

2.2.4 Componentes da SOA

SOA é um ambiente padronizado para os princípios da orientação a serviço. Neste ambiente é definido um conjunto de serviços que completam os objetivos e processos de negócios de uma

organização [Manolescu and Lublinsky, 2007a]. A seguir são descritos os principais componentes que fazem parte da SOA:

- **Mensagens** (unidades de comunicação): representa o dado solicitado para completar alguma ou toda parte de uma requisição;
- **Operações** (unidades de trabalho): representa a lógica solicitada para processar mensagens a fim de completar uma requisição;
- **Serviços** (unidades de processamento lógico): representa um conjunto de operações logicamente agrupadas capazes de executar as requisições relacionadas;
- **Processos** (unidades de automação lógica): contêm as regras de negócio que determinam quais operações de serviços são usadas para completar uma requisição.

Os componentes da SOA se relacionam da seguinte forma [Manolescu and Lublinsky, 2007a]:

- Uma operação envia e recebe mensagens para executar um trabalho;
- Uma operação é geralmente definida pelas mensagens que ela processa;
- Um serviço agrupa uma coleção de operações relacionadas;
- Um serviço é geralmente definido pelas operações que se relacionam com ele;
- Uma instância de processo pode compor serviços;
- Uma instância de processo não é definida necessariamente por serviços porque pode somente solicitar um subconjunto de funcionalidades oferecidas pelos serviços;
- Uma instância de processo invoca uma única série de operações para completar a automação;
- Cada instância de processo está parcialmente definida pelas operações de serviço que utiliza.

A figura 2.10 apresenta estes relacionamentos. As instâncias de processo executam uma série de operações, podendo também defini-las, e compõem um conjunto de serviços. Os serviços agrupam logicamente um conjunto de operações, e essas definem os serviços. As operações enviam e recebem mensagens, e essas definem as operações relacionadas.

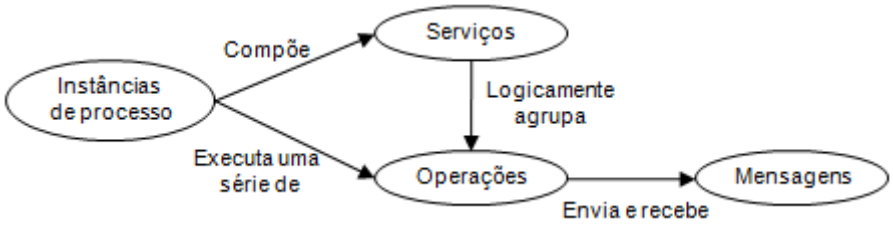

Figura 2.10 – Componentes de SOA relacionados [Manolescu and Lublinsky, 2007a].

2.2.5 Modelo da SOA

Um modelo de *software* implementado na SOA possui as seguintes características que são diferentes das arquiteturas de *software* tradicionais [Chen et al., 2006]:

- **Interoperabilidade baseada em padrões**: baseia-se em protocolos, comunicação, coordenação, *workflow*, descoberta, colaboração e publicação através de protocolos padrões tais como: XML, SOAP, WSDL, UDDI, HTTP, entre outros. Estes padrões permitem que os serviços desenvolvidos em diferentes plataformas possam interoperar uns com os outros conhecendo-se apenas a especificação do serviço;
- **Composição dinâmica via descoberta**: fornece uma nova forma de desenvolver aplicações baseadas em descoberta de serviços. Além disso, a composição e descoberta podem ser realizadas em tempo de execução;
- **Governança e orquestração dinâmica**: fornece mecanismos para controle da execução dos serviços. Um deles é um serviço de governança por políticas, ou seja, políticas podem ser especificadas, verificadas e reforçadas durante a fase de desenvolvimento e em tempo de execução. Outro mecanismo, chamado de orquestração, coordena a execução de um processo e é responsável por programar a execução de um serviço.

Com a introdução do conceito de serviços é estabelecida uma forma de abstração entre os processos de negócios e as aplicações já desenvolvidas nas organizações. A figura 2.11 apresenta um modelo de camadas de uma SOA, que atende as características mencionadas anteriormente, apresentando onde cada uma delas está representada.

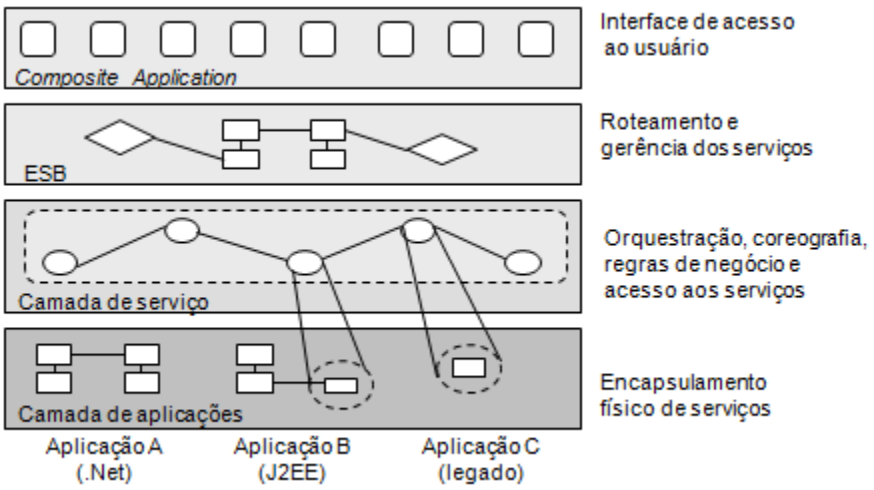

Figura 2.11 – Camadas da SOA [Erl, 2006].

Na figura 2.11, as aplicações A, B e C, pertencentes a uma camada de aplicações, distribuem seus dados como serviço para a camada de serviços, assim, a conexão com essas aplicações deixa de existir, restando apenas os serviços gerados a partir delas. A camada de serviços é responsável por gerenciar os processos de negócios, orquestrar e disponibilizar os serviços para o uso. A camada ESB efetua o roteamento e a gerência dos serviços disponibilizados e a camada de *composite application* fornece uma interface de acesso ao usuário para utilização do serviço requisitado.

2.2.5.1 Camada de serviço

No modelo de camadas apresentados na figura 2.11 são identificados quatro abstrações que compõem a camada de serviços, que são [Erl, 2006], [Chen et al., 2006]:

- **Acesso aos serviços**: são responsáveis por representar a tecnologia e a lógica de uma aplicação;
- **Regras de negócios**: são responsáveis por expressar a lógica do negócio através da orientação a serviços e trazer a representação de modelos de negócios corporativos para *Web Services*;
- **Coreografia**: pode ser vista como um processo que age de forma a permitir a colaboração entre diferentes orquestrações, ou ainda, podendo consistir de múltiplos participantes que podem assumir diferentes papéis e que possuem relacionamentos diferentes.

- **Orquestração**: representa o processo pelos quais diferentes serviços são invocados. Os serviços podem ser organizados em diferentes formas ou apenas reagrupados em outros fluxos. A orquestração é composta por um fluxo de etapas, e um coordenador responsável pelo andamento no fluxo.

  A figura 2.12 apresenta o processo de orquestração de serviços.

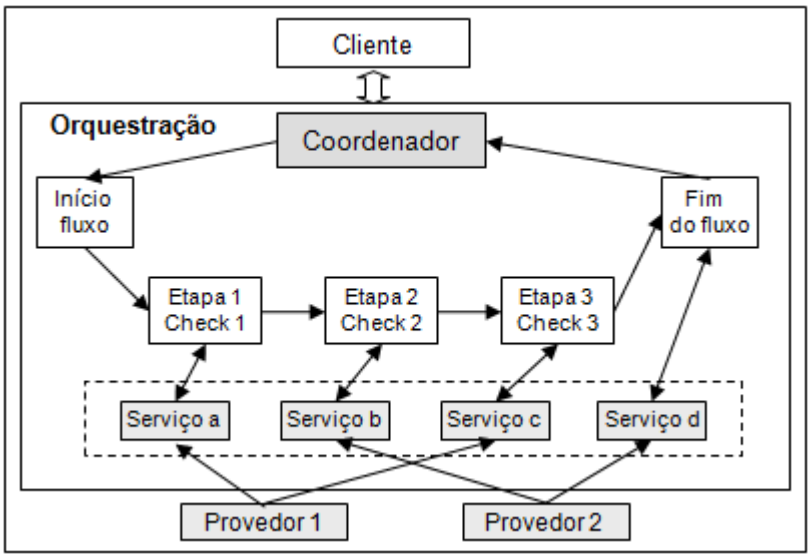

Figura 2.12 – Orquestração de serviços [Sampaio, 2006].

Na figura 2.12 é apresentado um esquema de orquestração de quatro serviços, fornecidos por dois provedores de serviços diferentes. Neste processo, o cliente se comunica com o coordenador e efetua uma solicitação. O coordenador inicia o fluxo, invocando e verificando todas as etapas necessárias. Cada etapa invoca um serviço, que é fornecido por um provedor de serviço. Desta maneira, é possível mudar a ordem das etapas, acrescentar outras, mudar os critérios de verificação ou criar outros fluxos sem alterar o código dos serviços.

A orquestração tem sido chamada de "o núcleo de SOA", estabelecendo um significado de centralização e controle das lógicas de negócios através de um modelo de serviço padronizado. A orquestração comanda um processo baseado em serviços e pode acessar diversas aplicações, implementar regras de negócios complexas e interagir com diferentes processos de negócios [Chappell, 2004].

Com a orquestração diferentes processos podem ser conectados redesenvolvendo as soluções originais. A orquestração introduz novas lógicas de fluxos de dados, assim, seu uso pode reduzir a complexidade das soluções estabelecidas. A lógica de fluxo de dados é abstraída e facilmente mantida [Chappell, 2004].

A camada de orquestração de serviços consiste de um ou mais serviços que compõem processos de negócios de acordo com a lógica de negócios. A orquestração abstrai as regras de negócios e a seqüência lógica de execução dos serviços, promovendo agilidade e reusabilidade.

2.2.5.2 *Enterprise Service Bus* (ESB)

De acordo com a IBM, um ESB ajuda a maximizar a flexibilidade de SOA. Os participantes de uma interação entre serviços são conectados ao ESB, e não um diretamente ao outro. Quando o serviço solicitante conecta no ESB, este assume a responsabilidade de entregar suas requisições, através do uso de mensagens, para o provedor de serviço que oferece a função solicitada [IBM, 2006].

O ESB facilita a comunicação entre solicitantes e provedores de serviços, resolvendo diferenças de protocolos, padrões de interação ou capacidade dos serviços. Também pode prover ou melhorar o monitoramento e gerenciamento sobre os serviços. O ESB provê virtualização e funcionalidades de gerenciamento que implementam e estendem as capacidades principais do SOA [IBM, 2006]. O ESB virtualiza a:

- **Localização e identidade**: participantes não precisam saber a localização ou identidade de outros participantes. Por exemplo, um serviço solicitante não precisa saber quais serviços podem atender sua requisição. Provedores de serviços podem ser adicionados ou removidos a qualquer momento sem problemas;
- **Protocolo de interação**: participantes não precisam usar o mesmo protocolo de comunicação ou modo de interação. Uma requisição feita por SOAP sob HTTP pode ser atendida por um serviço que apenas opere em SOAP sob JMS (*Java Message Service*);

- **Interface**: solicitantes e provedores não precisam utilizar uma interface comum. Um ESB concilia as diferenças transformando as requisições e respostas na forma esperada pelo serviço que irá recebê-las;
- **Qualidade de serviço** (QoS): participantes ou administradores de sistema especificam os requisitos de QoS, incluindo requisições de autorização, encriptação e decriptação de dados, auditoria de serviços e modo de roteamento de serviços (otimizando para velocidade ou custo, por exemplo).

A figura 2.13 apresenta como o ESB provê conectividade para estender diferentes pontos de SOA. Ambos os provedores e solicitantes de serviços conectam-se ao ESB para que a comunicação aconteça, conforme suas requisições.

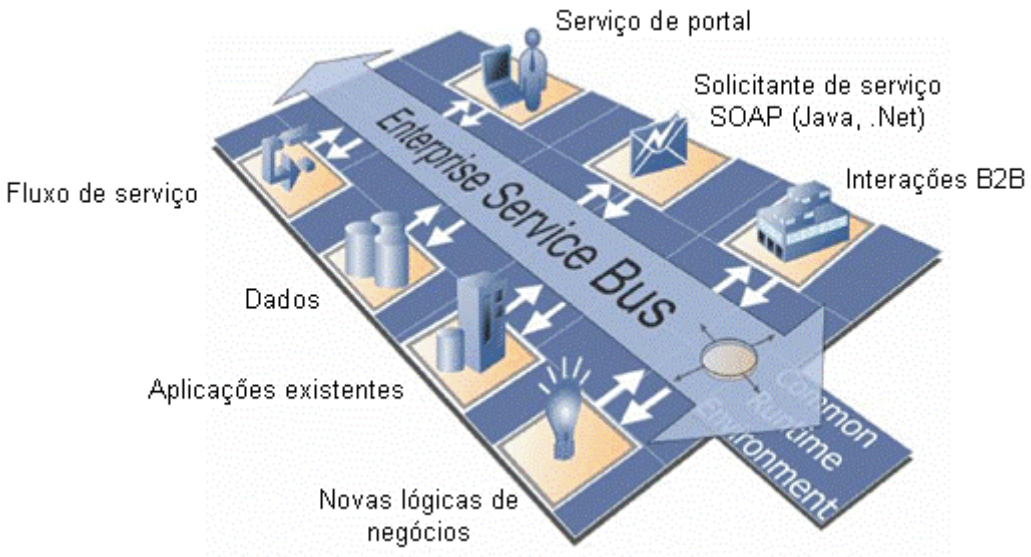

Figura 2.13 – Visão de infra-estrutura de um ESB [Keen et al., 2004].

2.2.5.3 *Composite Applications*

A globalização requer que as pessoas trabalhem de um modo mais colaborativo do que antes, tornando-se necessário, um intercâmbio entre as ferramentas utilizadas para adquirir compreensão, colaboração e que ajude na tomada de decisões. Hoje em dia, a maioria das aplicações de negócios são eficazes na automatização de transações, mas não permitem uma colaboração entre os limites funcionais. Isto usualmente leva pessoas a utilizarem ferramentas de produtividade para uso pessoal, por exemplo, um editor de texto, para realizarem suas

tarefas. Entretanto, isto também ocasiona uma perda de produtividade, pois são obrigados a mover-se de uma ferramenta para outra, trocando a informação de forma manual mediante os recursos como recortar e colar. Segundo Banerjee, estas diferenças entre as diversas aplicações de negócios e ferramentas de produtividade devem ser reduzidas para um modo mais fácil, sincronizado e seguro [Banerjee, 2007].

Uma composição refere-se à maneira de disponibilizar soluções na empresa, assemelhando-se a componentes pré-construídos. Isto inclui também habilidades de personalização e de customização, assim as pessoas podem facilmente e rapidamente modificar funcionalidades específicas da solução criada. Os benefícios são substanciais, porque a composição fornece meios para conseguir agilidade, adaptabilidade e alinhamento nos negócios da empresa [Banerjee, 2007].

Definições de *composite applications* são encontradas na literatura:

- Uma *composite application* é: "uma coleção de serviços que foram montados para fornecer uma potencialidade ao negócio. Estes serviços são artefatos que podem ser desdobrados independentemente, permitindo a composição e utilização das capacidades de plataformas específicas" [Banerjee, 2007].
- Schimidt define que uma *composite application* "é criada por um conjunto de serviços interconectados e parametrizados fornecidos por componentes através de, por exemplo, *Web Services*" [Schimidt, 2003].
- Segundo Crespo, uma *composite application* "é definida através da composição de vários serviços, onde o relacionamento entre eles será realizado por *roles* que fazem o papel de uma interface para a colaboração dos serviços" [Crespo, 2000].

A figura 2.14 apresenta um modelo simplificado do conceito de *composite application*, onde os usuários se conectam as *composite applications* disponibilizadas por meio da orquestração de diversos serviços.

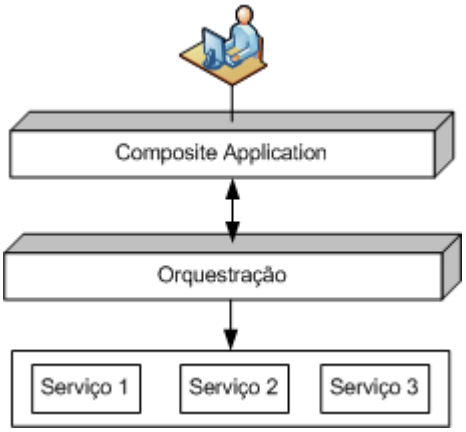

Figura 2.14 – Modelo de *composite application* [Banerjee, 2007].

Em SOA, *composite application* é o produto final. Estas representam o valor de negócio de uma empresa derivada da sua aplicação de SOA. Independente de a *composite application* ter sido planejada para uso interno ou externo, ela representa como uma empresa pode mapear suas necessidades e processos de negócios para que sejam disponibilizados através dos princípios de SOA.

Composite applications têm um grande potencial de mudar a maneira como as aplicações são construídas, entregues e utilizadas pelos usuários finais. Em alguns níveis, entretanto, isto complica o trabalho dos desenvolvedores de aplicações, pois se torna necessário considerar a experiência dos usuários no desenvolvimento. Quando um processo de negócio for disponibilizado como serviço, deve-se considerar com cuidado seus limites para que possa ser utilizado em ambientes compostos [Banerjee, 2007].

2.2.6 Arquitetura orientada a funcionalidades e arquitetura orientada a serviços

As diferentes arquiteturas existentes tornaram-se alvo de interessantes comparações com a proposta de SOA. Um estudo realizado por Erl identifica como SOA derivou de muitas características existentes em arquiteturas como: cliente-servidor, distribuída e *Web Services* [Erl, 2006].

SOA emprega tecnologias originalmente utilizadas para construir aplicações cliente-servidor, por exemplo, XML e *Web Services*. Entretanto, SOA e a arquitetura cliente-servidor diferem muito entre si. SOA elimina qualquer dependência com uma estação de trabalho de

usuário, delegando todo o processamento para dentro do servidor, contrário a arquitetura cliente-servidor.

A arquitetura distribuída da Internet tem muito mais em comum com SOA, incluindo as tecnologias de XML e *Web Services*. Entretanto, SOA tem características distintas para ambas as tecnologia e princípios de projeto. Por exemplo, SOA introduz requisitos de processamento e segurança que diferem da arquitetura distribuída da Internet, e a administração de SOA é tipicamente mais complexa no que se refere à comunicação baseada em mensagens.

Em relação à arquitetura orientada a objetos, não existe competição entre ambas, apenas comparação, visto que o projeto da arquitetura orientada a serviços está baseado no projeto de serviços e a arquitetura orientada a objetos está centrada na criação de objetos.

Muitos princípios de SOA são relacionados e derivados dos princípios da arquitetura orientada a objetos. O princípio de herança da orientação a objetos não faz parte dos princípios de SOA. Assim como, fraco acoplamento e autonomia, não são promovidos nos princípios da orientação a objetos. Na tabela 2.2 é apresentada uma lista de aspectos comparativos entre ambas as arquiteturas [Erl, 2006].

Tabela 2.2 – Comparação entre orientação a serviços e orientação a objetos [Erl, 2006].

Princípios da orientação a serviços	**Princípios da arquitetura orientada a objetos relacionados**
Serviços reusáveis	Abstração e encapsulamento suportam reuso, porém requerem uma distinta separação entre interface e implementação lógica.
Serviços compartilham contrato formal	O requisito de contratos de serviço é muito comparável para o uso de interfaces na construção de aplicações, mas ainda é considerada um dos itens de melhores práticas.
Serviços são fracamente acoplados	O uso de herança e outros princípios tornam os relacionamentos entre os objetos altamente acoplados.
Serviços devem abstrair lógica de negócios	Uma classe fornece uma interface para o mundo externo tornando-se acessível via interface.
Serviços são passíveis de composição	Suporte a conceitos de associação, tais como: agregação e composição.
Serviços são autônomos	Referências entre objetos e dependências relacionadas com herança suportam um pequeno grau de autonomia.
Serviços são *stateless*	Objetos combinam classes e dados sendo naturalmente *statefull*.
Serviços são passíveis de descoberta	Projetar classes de interfaces para serem consistentes e descritivas é um dos itens de melhores práticas.

2.3 Decomposição de serviço

Existe um termo comum nas definições de SOA encontradas na literatura que faz parte de seu conceito, ou seja, a forma de decompor os sistemas de *software* em artefatos menores, tornando-os mais gerenciáveis, para serem tratados como serviços, que vem a ser um dos requisitos essenciais de SOA. O processo de orquestração permite que estes serviços sejam

combinados para atender funcionalidades de negócios da empresa e após disponibilizá-los para uso.

A decomposição é uma técnica formalizada pela teoria clássica dos anos 50. Esta teoria enfatizava que quanto mais complexo um sistema é, mais desconhecido é seu conteúdo e assim mais difícil de automatizar. Esta teoria prescreveu a decomposição de sistemas complexos para sistemas menores, mais gerenciáveis, na qual é mais fácil de controlar e tratar o sistema como uma composição de suas partes. O mesmo se aplica as iniciativas de desenvolvimento de *software* complexos. Assim, a decomposição representa um dos pontos cruciais para uma boa engenharia de *software* [Parnas, 1972].

Manolescu e Lublinsky apresentam uma analogia a diferentes abordagens de decomposição de *software*. A primeira abordagem de decomposição de *software* surgiu nos anos 60, que idealizou a separação das aplicações em *jobs*, cada um implementado por um programa separadamente. Mais tarde, os programas das aplicações foram divididos em módulos e sub-rotinas de acordo com suas funções [Manolescu and Lublinsky, 2007a].

O paradigma orientado a objetos nos anos 70 introduziu no conceito de decomposição o modelo de objetos. Dando continuidade para criar um paradigma melhor de projeto, uma abordagem diferente foi introduzida nos anos 90, chamada componentes. A introdução de componentes de *software* melhorou a criação de aplicações tornando-as mais flexíveis, mais estruturadas e mais controláveis, porém considerando uma única aplicação. Hoje a proposta de SOA, baseada em serviços, é fornecer decomposição para os sistemas de uma empresa como um todo. A figura 2.15 apresenta uma abordagem da evolução da decomposição.

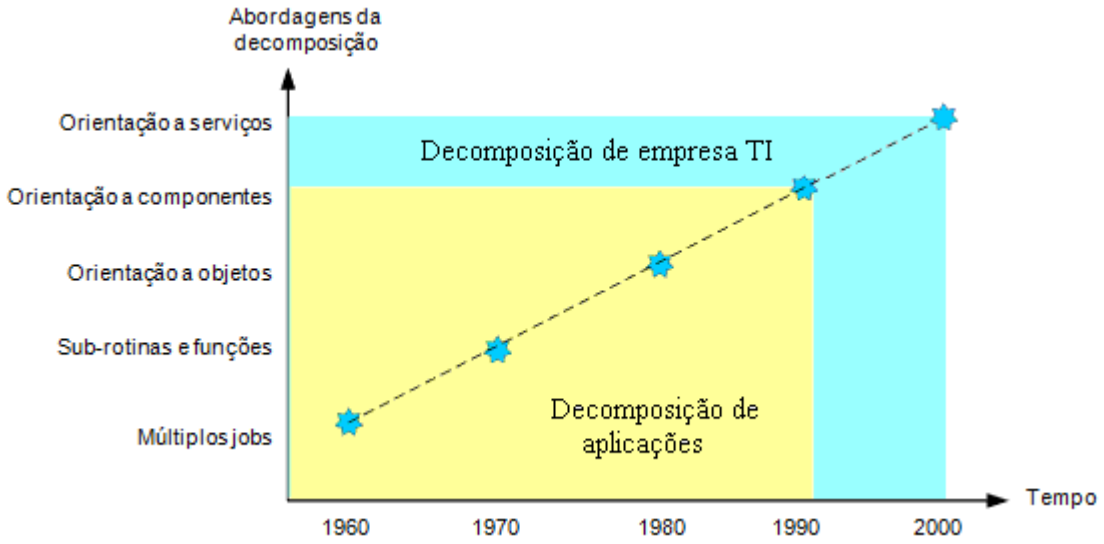

Figura 2.15 – A evolução da abordagem da decomposição [Manolescu and Lublinsky, 2007a].

Uma das questões mais difíceis para responder durante a implementação de SOA é como definir um serviço apropriadamente? Uma empresa usualmente já tem um conjunto de aplicações que suportam muitos dos requisitos de negócios funcionais. Estas aplicações são implementadas usando uma variedade de plataformas de *hardware*, sistemas operacionais e linguagens de programação. Algumas delas são implementadas pela empresa enquanto que outras são implementadas por parceiros de negócios [Manolescu and Lublinsky, 2007b].

A adoção de SOA [Manolescu and Lublinsky, 2007b] requer a definição de serviços que:

- Suportam as funcionalidades dos requisitos e processos de negócios;
- Alinhem as implementações de TI com funcionalidades de negócio;
- Aderem aos objetivos de arquitetura tais como: desempenho, escalabilidade, segurança.

O particionamento das funcionalidades de TI em serviços baseados no modelo de negócios da empresa e a refatoração deles para alinhar com os objetivos de arquitetura fornece uma solução para definir o modelo de serviços da empresa.

O processo de decomposição baseado em serviço possui as seguintes atividades, baseadas no modelo de negócios de empresa [Manolescu and Lublinsky, 2007b]:

- **Decomposição hierárquica**: assegura o alinhamento dos serviços;
- **Introdução de modelo semântico de dados**: assegura interoperabilidade dos serviços;
- **Refatoração dos serviços**: assegura a aderência aos objetivos de arquitetura.

A decomposição é uma técnica bem-estabelecida, e, dependendo do objetivo, muitos critérios de decomposição podem ser aplicados. O critério de decomposição tem um impacto significante nos objetivos de arquitetura tais como: desempenho, flexibilidade, compreensibilidade, tempo de desenvolvimento e reuso [Parnas, 1972].

2.4 Integração de aplicações

Segundo Erl, a motivação em busca de alguma forma de integrar aplicações esta direcionada para que duas ou mais aplicações consigam transferir dados. Esta transferência pode ser simples como, por exemplo, uma aplicação buscar um valor gravado em outro repositório de dados, ou intrínseca, como uma combinação ou colaboração de ambos os dados, recursos, e lógica de negócios das aplicações, para suportar os processos automatizados de outra aplicação [Erl, 2006].

Projetos de integração podem ser agrupados em duas categorias: aqueles que endereçam requisitos de negócios imediatos; e, aqueles que estabelecem um melhor caminho para permitir integração em geral. Assim, uma solução de integração pode ser direcionada para: estender uma aplicação existente, para suportar novas características ou uma modificação em um processo de negócio; ou, introduzir novos processos de negócios, criando uma nova dimensão de integração de aplicação, especialmente quando um processo requer que um grande número de aplicações tenha interoperabilidade [Kevin, 2006].

Alguns níveis de integração são descritos a seguir [Kevin, 2006]:

- **Integração a nível de dados**: uma aplicação pode acessar diretamente os dados de outra aplicação, ou o banco de dados de uma aplicação pode ser replicado para outra aplicação;

- **Integração a nível de aplicação**: uma aplicação faz uma solicitação de informação para outra aplicação envolvendo a lógica de negócios desta aplicação, e não o acesso ao banco de dados;
- **Integração a nível de processo**: o compartilhamento dos dados ou processos de negócios facilita a automatização destes processos, ou a combinação de dois ou mais processos existentes. Geralmente um componente chamado *broker* é utilizado para estabelecer a comunicação e, um componente de orquestração efetua o gerenciamento e execução de novos processos;
- **Integração orientada a serviços**: neste tipo de integração, um *Web Services* é utilizado como componente para estabelecer um padrão para compartilhamento de dados e lógica de programação. A figura 2.16 apresenta a integração orientada a serviços baseada em *Web Services*.

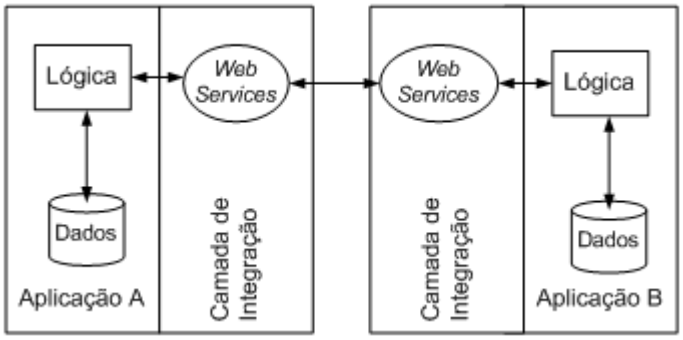

Figura 2.16 – Integração orientada a serviços com o uso de *Web Services*.

Na figura 2.16, tanto aplicação "A" quanto aplicação "B" ficam isoladas de qualquer acesso direto em seus códigos. Um *Web Services* criado na camada de integração é responsável por efetuar o acesso, a comunicação entre as requisições e efetuar a transferência dos dados solicitados.

A camada de integração criada para expor os dados da aplicação via *Web Services* é uma forma de adicionar componentes que podem ser utilizados efetivamente em uma variedade de arquiteturas. O nível de integração orientado a serviços estabelece uma fundamentação para um projeto de arquitetura de geração de serviços a partir de sistemas legados com o uso de *Web Services*, e são a base para o projeto da arquitetura utilizada nesta proposta.

2.5 Conclusão

A solução proposta consiste de *Web Services* no papel de serviços, mapeados para atender as necessidades de negócios já implementadas nos sistemas legados, ambientados num repositório qualquer na Web (podendo estar em diferentes máquinas, servidores de aplicação e plataforma) e comunicando-se via rede. Segundo os conceitos apresentados por Graham [Graham et al., 2002], pelo W3C [W3C, 2007] e por Erl [Erl, 2006], optou-se por "disponibilizar" *Web Services* sob o conceito de serviços, de forma que uma regra de negócio possa ser definida, descrita e descoberta. As características incorporadas ao uso de *Web Services* serão apresentadas no capítulo 4, onde a arquitetura Aruba é descrita.

Os *Web Services* utilizarão XML como base de sua comunicação, seguindo o conceito já apresentado por Erl como requisito essencial para comunicação em SOA [Erl, 2006]. O estudo realizado, incluindo a análise de decomposição de serviços, serviu como base para a geração dos serviços a partir das regras de negócios existentes nos sistemas legados. Já o estudo de diferentes formas de integração de aplicações, serviu para definir que a Aruba utilizará a integração orientada a serviços e a necessidade de disponibilizar estes serviços levou à definição do uso de *Web Services*, que seguirão a definição dos princípios da orientação a serviços descritos em Erl.

O capítulo a seguir apresenta trabalhos relacionados com o trabalho proposto. Alguns destes são ferramentas que contemplam outras funcionalidades da SOA incluindo orquestração, ESB e *composite application* que não fazem parte da proposta deste trabalho.

3 TRABALHOS RELACIONADOS

A geração de serviços na forma de *Web Services* não é algo novo, visto que sua utilização já possui diversos guias para implementação. Este capítulo busca os principais trabalhos relacionados a esse tema e que trazem contribuições para solucionar alguns problemas já encontrados, além de servirem para posicionar o presente trabalho em relação a outros na mesma área. O capítulo apresenta uma solução de migração de sistemas legados para SOA denominada SMART, que não contempla questões de codificação. Logo após, é apresentada uma técnica para encapsulamento de sistemas legados para reuso em SOA, que propõe três passos para a geração de serviços. Por fim, apresenta quatro ferramentas comerciais que permitem geração de serviços, além de outras funcionalidades não contempladas neste trabalho.

3.1 Migração de sistemas legados para SOA

Para Lewis, tornar possível que um sistema legado trabalhe com *Web Services* é às vezes relativamente simples. As interfaces de *Web Services* são baseadas em padrões bem conhecidos e são configuradas para receber mensagens, transformar seu conteúdo, fazer uso de código legado, e opcionalmente alterar os resultados das mensagens retornadas ao fornecedor [Lewis et al., 2005b]. Porém, existem características dos sistemas legados que podem tornar o processo de criação de *Web Services* uma tarefa complicada, por exemplo:

- Tempo de existência;
- Linguagem de programação;
- Arquitetura;
- Estado futuro baseado em serviços.

Os autores propõem uma técnica que ajuda na análise de sistemas legados de organizações a fim de determinar quais funcionalidades, ou subconjuntos destas, podem ser expostas como serviços para uso em uma SOA. A idéia central está baseada em cinco atividades que devem ser desenvolvidas, sendo que, cada uma destas, está dividida em muitas outras tarefas. A técnica fornece uma análise preliminar da viabilidade da migração de componentes legados para serviços, disponibiliza estratégias de migração, incluindo os custos e riscos envolvidos [Lewis et al., 2005a].

A técnica SMART (*Service-Oriented Migration and Reuse Technique*) será apresentada na próxima seção.

3.1.1 A técnica SMART

O SMART foi derivado do método *Options Analysis for Reengineering* (OAR) desenvolvido pelo *Software Engineering Institute* (SEI) utilizado para suporte na análise e reuso de sistemas legados [Bergey, 2002]. Como mencionado anteriormente, a técnica SMART está dividida em cinco atividades principais, representadas na figura 3.1. As atividades de entrada são providas de documentações e entrevistas com as pessoas envolvidas da organização, e as atividades de saída são o resultado das atividades desenvolvidas.

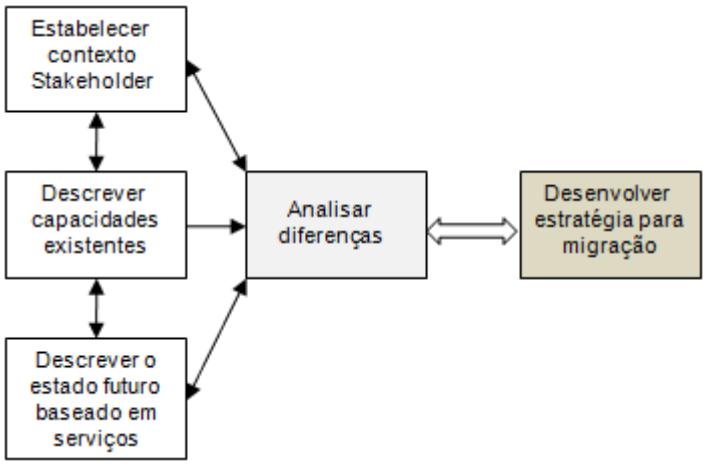

Figura 3.1 – Atividades do SMART [Lewis et al., 2005a].

A busca pelas informações para as três primeiras atividades são direcionadas pelo *Service Migration Interview Guide* (SMIG). O SMIG contém questões que diretamente endereçam as diferenças entre a arquitetura existente e a que se pretende atingir, o projeto, e o código, assim como questões que devem ser endereçadas aos esforços da migração para serviços. As cinco atividades destacadas na figura 3.1 são descritas a seguir.

1. **Estabelecer o contexto com os *stakeholders***: atividade que identifica características sobre o sistema legado, seu objetivo atual, e o que deverá ser migrado para serviço ou conjunto de serviços. Informações sobre o objetivo da migração, expectativas, usuários potenciais de serviços, usuários que utilizam o sistema legado e os envolvidos no processo de migração são identificadas nesta atividade;

2. **Descrever as capacidades existentes**: o objetivo desta atividade é obter dados descritivos sobre os componentes do sistema legado. Dados como o nome, função, tamanho, linguagem de programação, sistema operacional e tempo de existência dos componentes legados são solicitados. Responsáveis técnicos são questionados sobre a arquitetura, paradigma de projeto, complexidade do código, nível de documentação, acoplamento dos módulos, interfaces para sistemas e usuários, dependências com outros componentes e produtos comerciais;
3. **Descrever o estado futuro baseado em serviços**: o objetivo desta atividade está em obter evidências sobre os serviços potenciais que podem ser criados dos componentes legados; e, obter detalhes suficientes sobre o que se almeja com SOA para suporte nas decisões sobre que serviços podem ser apropriados e como eles interagem uns com os outros e o SOA;
4. **Analisar as diferenças entre o estado baseado em serviço e as capacidades existentes**: o objetivo desta atividade é identificar a diferença entre o estado existente e o futuro e determinar o nível de esforço e custo necessários para converter os componentes legados para serviços. Recursos opcionais podem ser utilizados para capturar informações mais detalhadas sobre o sistema legado, tais como: uso de analisador de código e ferramentas de reconstrução de arquitetura para analisar o código fonte existente. Em muitos casos, a documentação torna-se insuficiente ou não há certezas sobre características como dependências com produtos comerciais;
5. **Desenvolver uma estratégia para migração de serviços**: a atividade final do SMART inclui identificar os componentes específicos para migrar, planejar e ordenar os esforços para a migração, caminhos específicos a serem seguidos (alterações simples versus reescrita do código), identificação das melhorias que levam ao aumento das capacidades, atribuição da melhor equipe para liderar a responsabilidade da migração e sugerir uma coordenação com os esforços relacionados, por exemplo, a construção da infra-estrutura SOA.

Pode-se perceber que a técnica SMART propôs um modelo para expor as funcionalidades de sistemas legados como serviços, porém, esta técnica está muito direcionada em gerar informações que justifiquem a viabilidade deste objetivo.

3.2 Encapsulando *softwares* legados para reuso em SOA

Em Sneed [Sneed, 2006] é apresentado um método que ajuda na identificação de quais partes de um sistema legado poderiam ser expostas como *Web Services* e uma técnica de como fazer esta transformação. Esta solução está baseada num método disponibilizado por uma ferramenta, que permite integrar o código legado dentro de uma interface XML em funções separadas, para serem oferecidas como *Web Services* para algum usuário externo. O foco deste método consiste de três passos básicos para a criação de *Web Services* a partir de sistemas legados:

1. **Preservando o código legado**: neste primeiro instante, é sugerido que seja realizada uma análise do código legado existente para identificar qual parte do código agrega valor para ser reutilizado. Esta análise pode ser realizada por algum programador que esteja familiarizado com o código ou ainda sugere-se o uso de alguma ferramenta de engenharia reversa. Com o resultado desta análise, blocos de códigos do sistema legado são copiados juntamente com as variáveis que estão referenciadas como entrada e saída de dados. Esta técnica é chamada de "*Code Stripping*" [Sneed, 1996], aplicada em tarefas de extração de operações de negócios elementares. Com o código legado identificado, o próximo passo é extrair o código e reagrupá-lo em módulos separados com interfaces próprias. As unidades de código são copiadas para um *framework* e os objetos de dados que são referenciados por eles são agrupados em uma interface de dados comum. Os argumentos de entrada e saída originais são os parâmetros de entrada e saída atuais. Neste passo, o código da lógica de negócio é desconectado da interface original com o usuário e feito um subprograma independente;

2. **Encapsulando o código legado**: neste passo, o objetivo é fornecer um componente extraído do código legado através de uma interface WSDL. A técnica utilizada consiste de transformar cada entrada para um método e transformar cada parâmetro em um elemento de dado XML. As estruturas de dados tornam-se elementos complexos com um ou mais sub-elementos. Os métodos terão seus argumentos e os resultados como referências para as descrições dos elementos de dados. Ambos os métodos e parâmetros são construídos no XML *Schema*. A figura 3.2 apresenta o processo de encapsulamento;

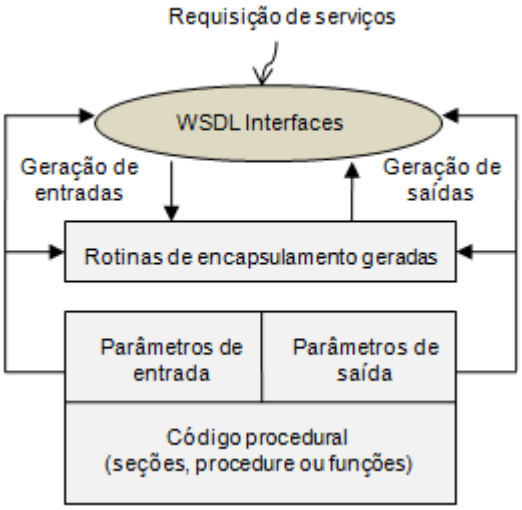

Figura 3.2 – Encapsulando componentes preservados [Sneed, 2006].

3. **Tornar o código disponível como um *Web Services***: o último passo desta solução diz respeito a fazer um *link* do *Web Services* com o processo de negócio. Isto é feito por meio de um componente de *proxy*. O processo de negócio invoca o *proxy* que esta disponível no mesmo espaço de endereço que as de definição do processo. O *proxy* verifica os parâmetros e gera interfaces WSDL que são despachadas por algum serviço de mensagem, tal como, *MS-Series* para o *application server*. Assim, o *application server* recebe uma mensagem, determina qual *Web Services* será executado e envia o conteúdo WSDL para este serviço em particular, neste caso o código legado encapsulado. O encapsulador do código efetua uma tradução dos dados de entradas XML e move os valores para os endereços apropriados no componente encapsulado. O resultado é transformado pelo encapsulador em uma estrutura de dados de saída XML, para ser transmitida novamente ao *Web Client*. Desta forma, os processos de negócios podem ser executados por qualquer cliente em qualquer local e podem ainda acessar as funções legadas do *application server* original.

Resumidamente, este trabalho propõe um método para identificar quais funções de negócios serão disponibilizadas como *Web Services*. O código legado é preservado, porém a lógica de negócios que será disponibilizada deve ser extraída e salva em outro arquivo, de forma que todas as funções dependentes sejam juntamente agrupadas em uma única função seqüencialmente. Após isso, são identificados os parâmetros de entrada e saída, gerado um XML capaz de fazer a tradução destes parâmetros e por fim, um *proxy* é utilizado para fazer a comunicação entre o cliente e o código legado alterado.

Com relação a este trabalho, um ponto importante identificado é que o código do sistema legado sofre modificações para torná-lo disponível como *Web Services*. Um bom conhecimento da lógica de negócios implementada é necessário para efetuar a extração das funções para serem reagrupadas novamente em um único arquivo.

3.3 Sistemas legados e SOA: Soluções comerciais

Foram encontradas ferramentas comerciais que fazem uso de *Web Services* para expor as funcionalidades de sistemas legados para uso em SOA, entre elas estão: IBM *SOA Foundation*, Microsoft *BizTalk Server*, Oracle *SOA Suite* e SAP *NetWeaver*. Cada uma destas ferramentas apresenta uma abordagem diferente ou proprietária para geração de *Web Services*, possuem diferentes mecanismos para desenvolvimento SOA e fornecem soluções desde integração com sistemas legados até a disponibilidade dos serviços para uso por outras aplicações.

3.3.1 IBM *SOA Foundation*

Segundo a IBM, SOA fornece flexibilidade e reuso dos processos de negócios de uma empresa. SOA propõe uma arquitetura que combina conexões adaptáveis com relações bem definidas, baseadas em tecnologias padrão para ajudar a flexibilizar infra-estruturas existentes [IBM, 2005].

Os serviços de SOA são extensíveis, baseados na implementação de novos serviços ou recursos de TI existentes. Assim, o desenvolvimento de uma SOA começa com uma infra-estrutura flexível, robusta que pode ser utilizada em conjunto com outra infra-estrutura existente e recursos de TI para proporcionar mais valor ao negócio.

A IBM definiu juntamente com seus clientes o ciclo de vida de SOA, apresentado na figura 3.3 a seguir [IBM, 2005]:

Figura 3.3 – Ciclo de vida de SOA proposto pela IBM [IBM, 2005].

- **Modelo**: levantamento dos requisitos de negócios e projeto dos processos de negócios;
- **Construção**: para os processos otimizados, são construídos componentes que são combinados aos serviços existentes para dar forma aos processos de negócios;
- **Implantação**: os processos otimizados são implantados num ambiente de serviços altamente seguro e integrado;
- **Controle**: a informação levantada durante a fase de controle é realimentada para o ciclo de vida para permitir melhoria contínua no processo;
- **Governança e processos**: fornecem orientação e visão gerencial para o projeto SOA.

A plataforma IBM *SOA Foundation* foi desenvolvida para atender todas as camadas da arquitetura de referência SOA da IBM, na qual define serviços de TI requeridos para fornecer suporte a cada um dos estágios do ciclo de vida de SOA apresentados anteriormente. A arquitetura de referência SOA inclui um ambiente de desenvolvimento, gerenciamento de serviços, integração de aplicações e processo de serviços em tempo de execução [IBM, 2005].

A figura 3.4 apresenta a arquitetura de referência SOA da IBM.

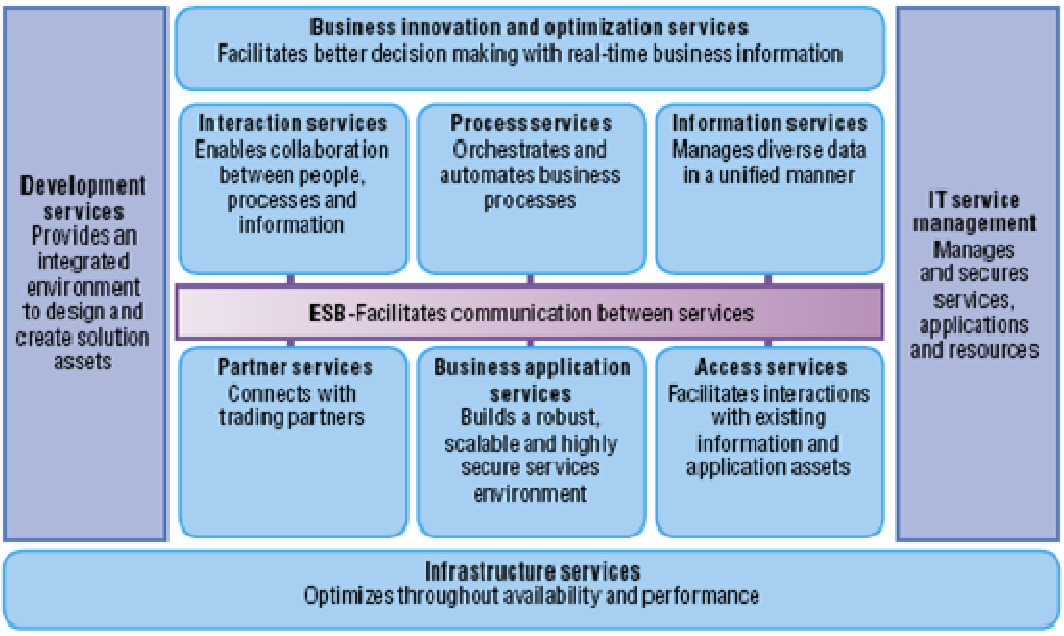

Figura 3.4 – Arquitetura de referência SOA da IBM [IBM, 2005].

Para conseguir abranger toda a arquitetura de referência, a plataforma IBM *SOA Foundation* está dividida em diversas ferramentas, cada uma atendendo uma função específica na arquitetura [IBM, 2006]. Dentre elas algumas se destacam na criação de *Web Services*, que serão apresentadas a seguir:

- IBM *Rational® Software Architect*: uma vez criado o processo de negócio e mapeados os serviços, essa ferramenta permite ao arquiteto de *software* definir quais serão as fontes e origens dos dados dos serviços pré-definidos pelo analista de negócios;
- IBM *Rational Application Developer*: ferramenta utilizada por desenvolvedores. Com esta ferramenta é possível criar código J2EE[10], por exemplo, *servlets* e *Enterprise Java Beans* (EJB), testar o código e implantar no servidor em tempo de execução. Com as ferramentas de *Web Services* é possível criar *Web Services* consumidores, *Web Services* provedores e mapear aplicações existentes como serviços. É um ambiente de desenvolvimento que auxilia na programação de interfaces a partir dos serviços existentes. Estas interfaces são disponibilizadas para os usuários;
- IBM *WebSphere Integration Developer*: ferramenta utilizada por arquitetos. Contém vários adaptadores para diferentes bancos de dados ou sistemas de gestão, com o

[10] Informações podem ser encontradas em: <http://www.sun.com>

objetivo de permitir a conexão dos serviços com qualquer fonte de dados. É possível criar um processo de negócios com *Business Process Execution Language* (BPEL), configurar adaptadores, criar mediadores, testar o código e implantar no servidor em tempo de execução.

3.3.2 Microsoft

Segundo a Microsoft, a orientação a serviços é uma abordagem para organizar recursos distribuídos de TI em uma solução integrada que desmembra grupos de informação e maximiza a organização na agilidade dos negócios. Os serviços se comunicam entre si por meio de formatos de mensagens bem-definidos; isso significa que a confiabilidade do aplicativo de sistemas conectados sofrerá uma grande influência da confiabilidade da infra-estrutura de mensagens que ele usa para fazer a comunicação entre os serviços. A orientação a serviços une fontes de informações autônomas construindo uma ponte em grande escala de sistemas operacionais, tecnologias, e protocolos de comunicação [Microsoft, 2006a].

Um processo em SOA ocorre de forma iterativa iniciando pela exposição de novos serviços, composição destes serviços em *composite applications*, e disponibilização para consumo por usuários ou outras aplicações de negócios. A figura 3.5 apresenta esta visão do ciclo de vida de SOA.

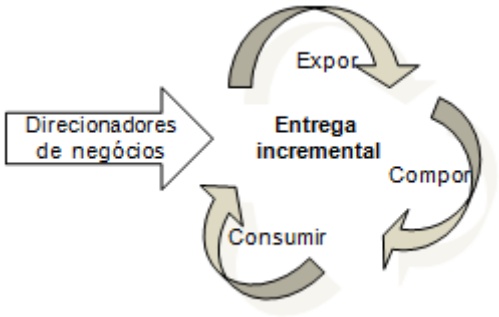

Figura 3.5 – Ciclo de vida de SOA proposto pela Microsoft [Microsoft, 2006a].

Os componentes do ciclo de vida apresentado da figura 3.5 são descritos a seguir:

- **Expor**: foco nos serviços a serem criados das aplicações e dados em questão;
- **Compor**: serviços criados são combinados com outros serviços, aplicações, ou outros processos de negócios;

- **Consumir**: os serviços criados são disponibilizados para uso por outras aplicações.

Entre alguns aplicativos da Microsoft que ganham destaque na colaboração para desenvolvimento de aplicações voltadas para SOA estão [Microsoft, 2006b]:

- **Microsoft *BizTalk Server* 2006**: pode-se dizer que no ambiente Microsoft, o *BizTalk Server* é a plataforma para desenvolver SOA. É um servidor da Microsoft que permite que os clientes integrem sistemas, funcionários e parceiros comerciais e reúne as funcionalidades de integração de aplicativos e automação de processos das tecnologias XML e *Web Services*. O *BizTalk Server* funciona como um mecanismo de execução de processo e como um *hub* de sistema de mensagens, fornece suporte ao consumo de serviços da Web como parte de um processo comercial, expondo os processos comerciais e aplicativos de linha de negócio como serviços da Web. Ele fornece suporte a SOAP, UDDI, WSDL entre outros, utilizando adaptadores ASMX e *Web Services Enhancements* (WSE). Ele também acrescenta a capacidade de chamar serviços da Web por meio de serviços de mensagens públicas/subestilo e fornece um adaptador *Windows Communication Foundation* (WCF) para incorporar serviços da Web WCF a processos comerciais. A figura 3.6 apresenta como o *BizTalk Server* facilita a exposição de processos comerciais como serviços da Web;

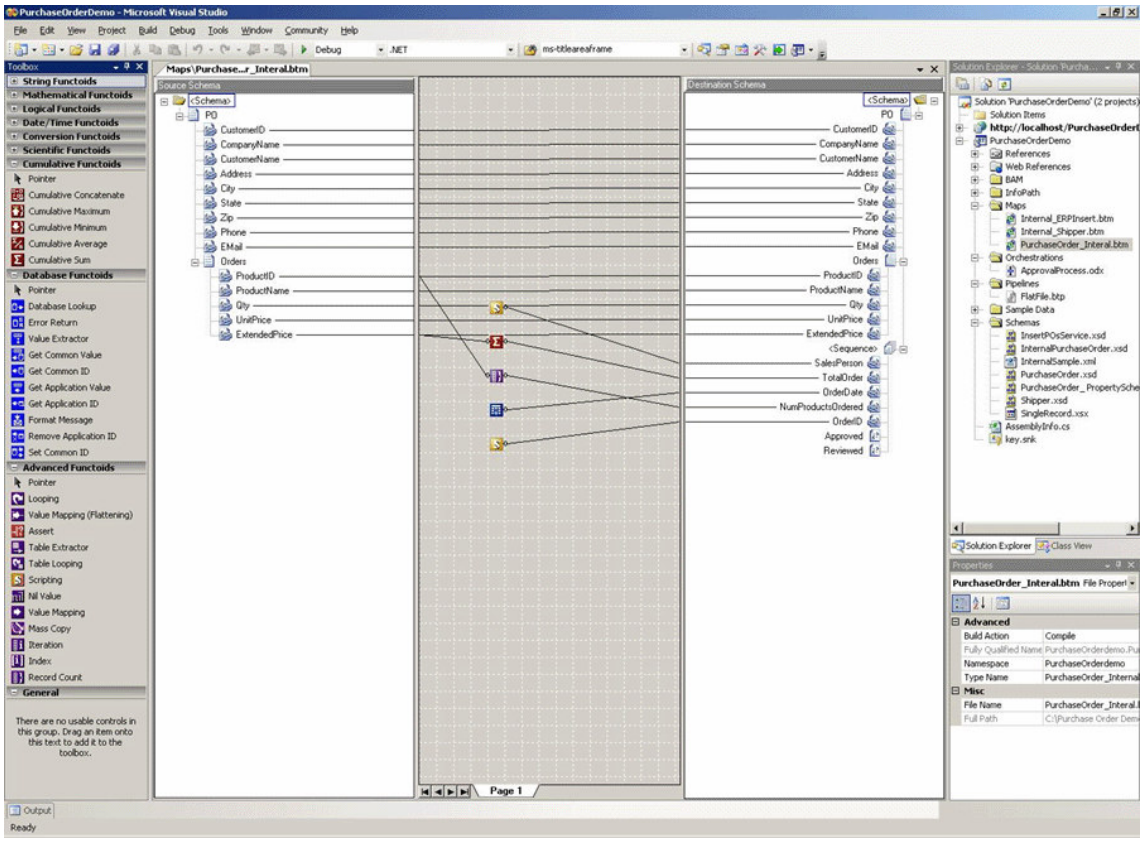

Figura 3.6 – Exposição de processos de negócios como serviços da Web.

- **Microsoft *Visual Studio***: é o ambiente de desenvolvimento de aplicativos totalmente gerenciado. Permite o desenvolvimento de aplicações Web e cliente/servidor, assim como, criação de serviços. O Visual Studio gera automaticamente um código *proxy* para serviços da Web definidos por WSDL, permitindo a utilização de serviços em aplicativos do Microsoft *Windows*, da Web, móveis e baseados no Microsoft *Office*. A publicação e localização de serviços em UDDI também são facilitadas por este ambiente. A figura 3.7 apresenta a interface do *Visual Studio* que fornece suporte a geração automática do código *proxy* a partir de WSDL.

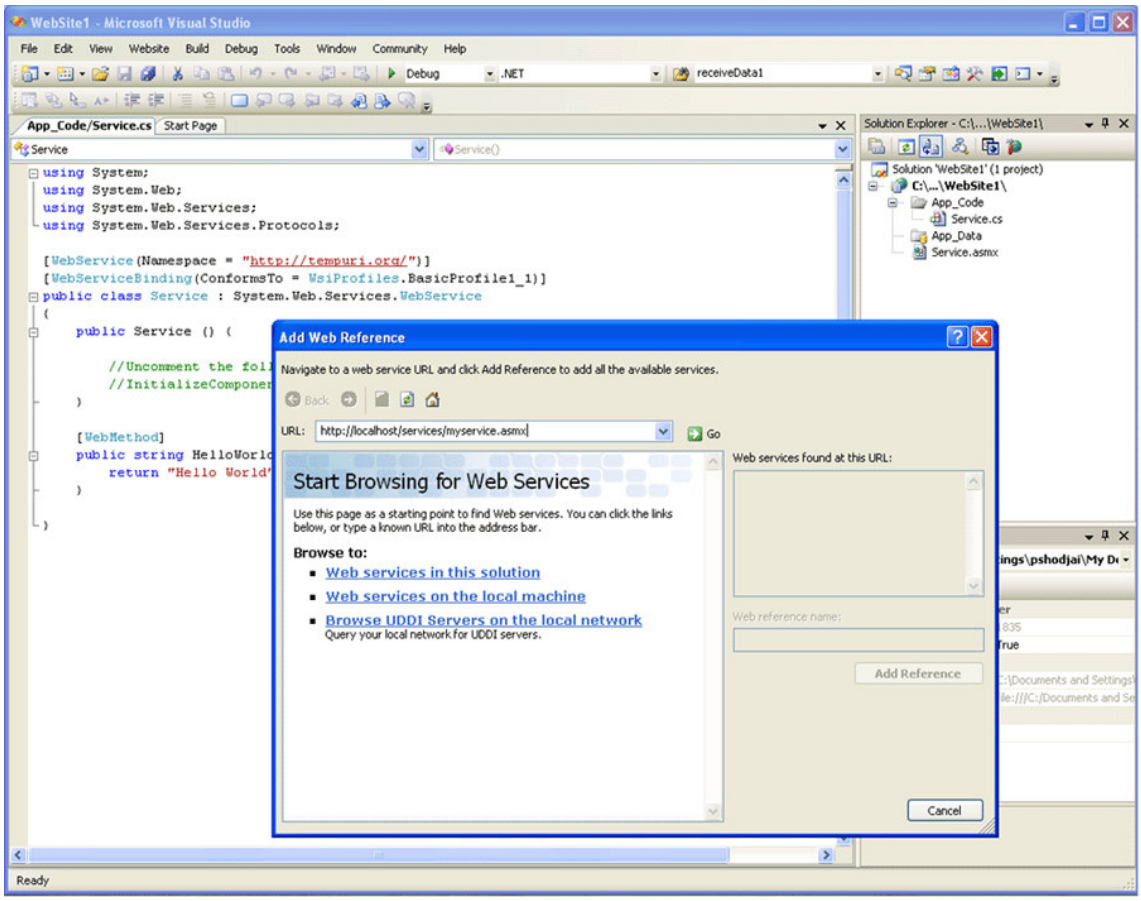

Figura 3.7 – Caixa de diálogo *Add Web Reference* do *Visual Studio*.

As soluções Microsoft não estão baseadas em padrões abertos como BPEL, apenas são compatíveis. Quando as ferramentas da Microsoft são utilizadas para SOA, por padrão são utilizados padrões proprietários da empresa.

3.3.3 Oracle *SOA Suite*

Segundo a Oracle, muitas empresas estão direcionando a complexidade de suas aplicações e ambientes de TI para SOA. SOA fornece uma arquitetura que suporta a construção de aplicações conectadas entre empresas, facilitando o desenvolvimento dessas aplicações como *Web Services* que podem ser facilmente integrados e reutilizados, criando uma infra-estrutura de TI flexível e adaptável [Oracle, 2006].

A Oracle publicou o ciclo de vida de SOA. O ciclo apresenta como está organizado o processo de implementação de SOA e é composto de uma seqüência de passos, incluindo

desde o desenvolvimento inicial dos serviços, seqüenciamento, gerenciamento até sua utilização [Oracle, 2005]. A figura 3.8 apresenta os sete estágios do ciclo de vida de SOA:

Figura 3.8 – Ciclo de vida de SOA [Oracle, 2005].

- **Desenvolver**: projeto e construção de serviços que correspondem às etapas específicas dentro de um processo de negócio;
- **Integrar**: integração dos serviços construídos com outros serviços, sistemas, banco de dados, que requer transformações de dados para mapeamento entre diferentes esquemas de dados, assim como rotinas dinâmicas para conectar os serviços apropriados em tempo de execução;
- **Orquestrar**: com os serviços desenvolvidos, a orquestração fornece o seqüenciamento dos serviços a fim de combinar tarefas ou processos de negócios;
- **Assegurar**: antes dos serviços serem implantados, o acesso para eles deve ser definido;
- **Controlar**: definir e reforçar acordos em nível de serviço, e políticas operacionais para auditar e faturar (se necessário) o uso do serviço;
- **Acessar**: os serviços são expostos através de um portal ou uma *composite application*;

- **Analisar**: a análise dos serviços, eventos e processos de negócios envolvidos em operações de negócios geralmente necessita ocorrer em tempo real, tornando possível o monitoramento, análise e resposta pelos gerenciadores.

Assim como a IBM, a Oracle oferece uma plataforma completa, chamada Oracle *SOA Suite*, para a adoção de SOA, desde a especificação até a implantação, além de ferramentas para manutenção, melhoria e gerenciamento da arquitetura. Todas essas ferramentas trabalham com padrões abertos e são integráveis com ferramentas de outros fornecedores [Oracle, 2006].

O Oracle *SOA Suite* é o conjunto de ferramentas da Oracle para criação de aplicativos sob o conceito de SOA assim como adaptação e transformação de aplicativos existentes em serviços [Oracle, 2006]. A figura 3.9 apresenta um diagrama com os principais processos da plataforma.

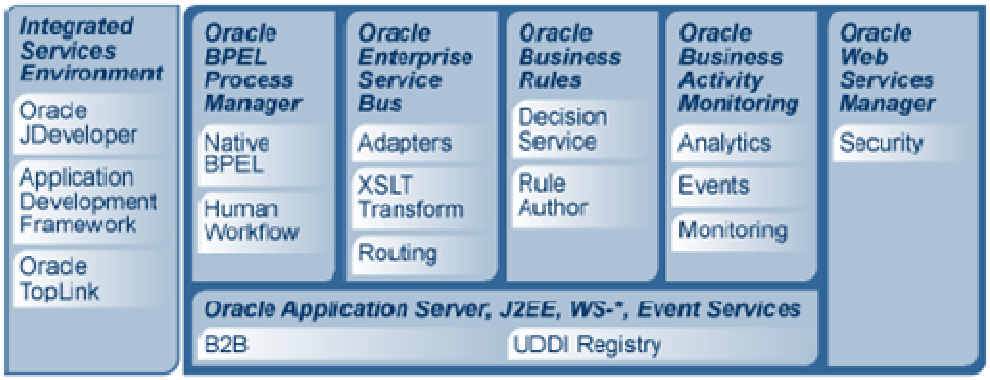

Figura 3.9 – Diagrama dos processos do Oracle *SOA Suite* [Oracle, 2006].

Dentre os componentes que fazem parte do Oracle *SOA Suite*, um deles será descrito a seguir, pois está relacionado com a criação de *Web Services* a partir de sistemas legados.

O Oracle *JDeveloper* é um ambiente de desenvolvimento da Oracle que é utilizado também para programar *Web Services* para serem inclusos como serviços no catálogo ou desenvolver na linguagem Java *composite applications* que farão uso dos serviços disponíveis no ambiente. A ferramenta de desenvolvimento deve estar instalada e configurada na máquina cliente para uso.

O processo de criação e disponibilização de um serviço no Oracle *JDeveloper* funciona da seguinte forma:

1. Primeiramente verifica-se se o arquivo .Java do sistema legado está disponível para utilização, caso não encontra-se, deve-se utilizar outro programa que faça a conversão deste arquivo. A figura 3.10, apresenta a interface do *Software Cavaj* utilizado para gerar o arquivo .Java a partir do arquivo .class.

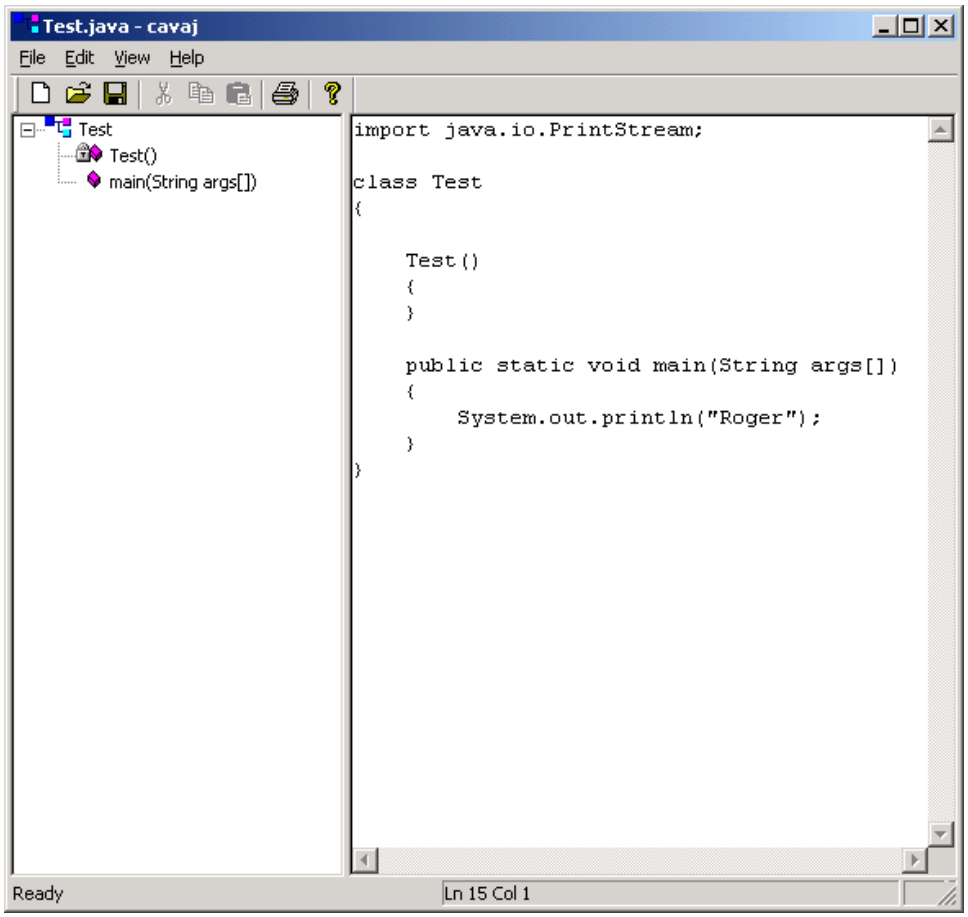

Figura 3.10 – Geração do arquivo .Java com Cavaj.

2. O próximo passo é criar um projeto a partir do arquivo gerado e mapear todas as suas dependências com as classes existentes. A figura 3.11 apresenta a interface do Oracle *JDeveloper*.

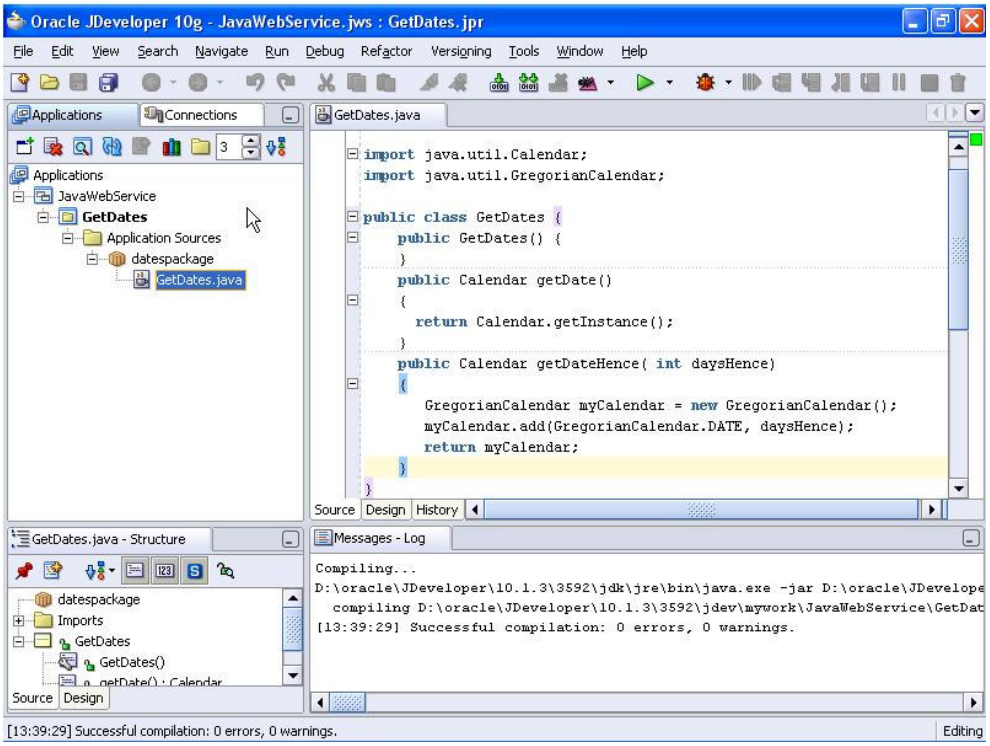

Figura 3.11 – Interface do Oracle *JDeveloper* de mapeamento.

3. A partir do arquivo .Java criado e suas dependências mapeadas, utilizar uma Wizard para criação do *Web Services* e arquivo do WSDL, a figura 3.12 apresenta as interfaces correspondentes no Oracle *JDeveloper*.

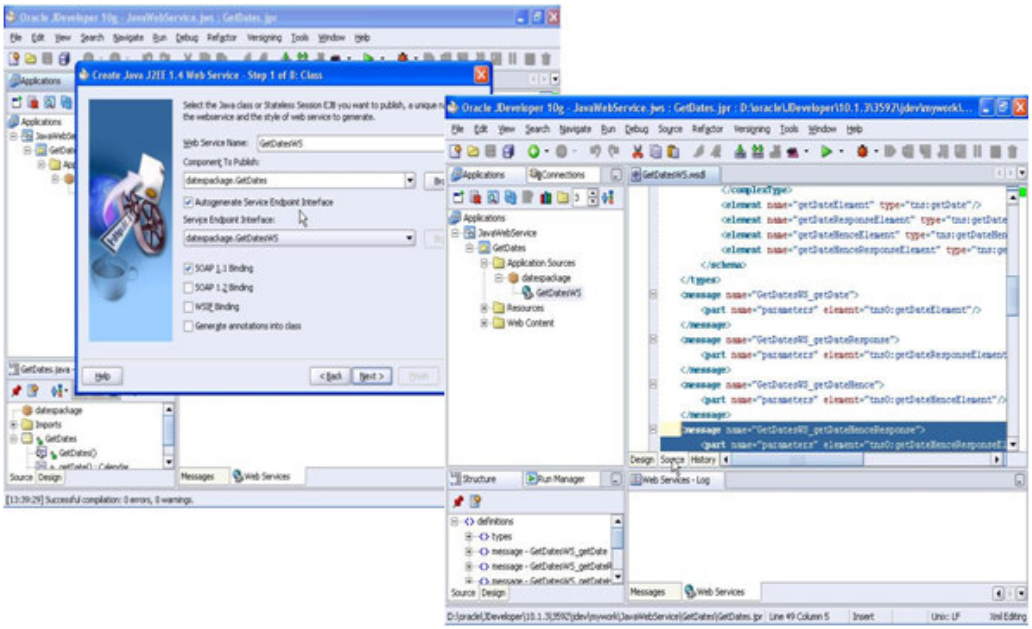

Figura 3.12 – Interface do Oracle *JDeveloper* para criação de *Web Services*.

4. O próximo passo refere-se a geração de um arquivo chamado *deployment profile* que serve para configurar a URL e o *application server* para utilização. Após gerado, o arquivo poderá ser executado. A figura 3.13 apresenta as interfaces correspondentes no Oracle *JDeveloper*.

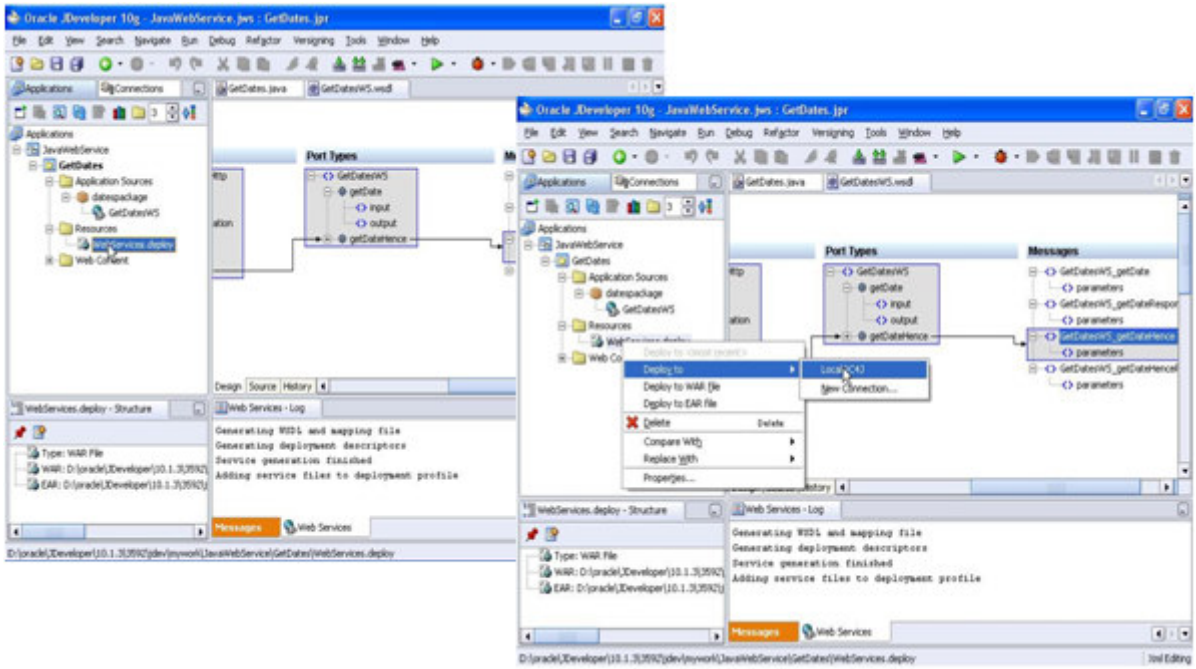

Figura 3.13 – Interface do Oracle *JDeveloper* para configuração do *Web Services*.

3.3.4 Solução da SAP para SOA

A SAP é uma empresa mundial e consagrada na área de *Enterprise Resource Planning* (ERP). Sua estratégia para SOA é encapsular os conceitos desta arquitetura sob a sua plataforma SAP *NetWeaver*, ou seja, ao contrário das outras empresas que oferecem soluções parciais e integráveis que juntas formam uma plataforma para desenvolvimento, uso e gerenciamento do SOA, a SAP criou sua própria solução, denominada de *Enterprise Services Architecture* (ESA), com o intuito de que os sistemas legados se convertam a essa arquitetura [SAP, 2006]. Na prática pode-se dizer que a arquitetura ESA, em si, não é mais do que um nome comercial da SAP para a SOA. Sendo assim, a diferença prática entre ESA e SOA resume-se ao uso ou não da plataforma SAP.

O SAP *NetWeaver* é uma plataforma voltada para integração, colaboração e interação entre pessoas, informações, processos e aplicações [Heuser, 2004]. A figura 3.14 apresenta um overview do SAP *NetWeaver*.

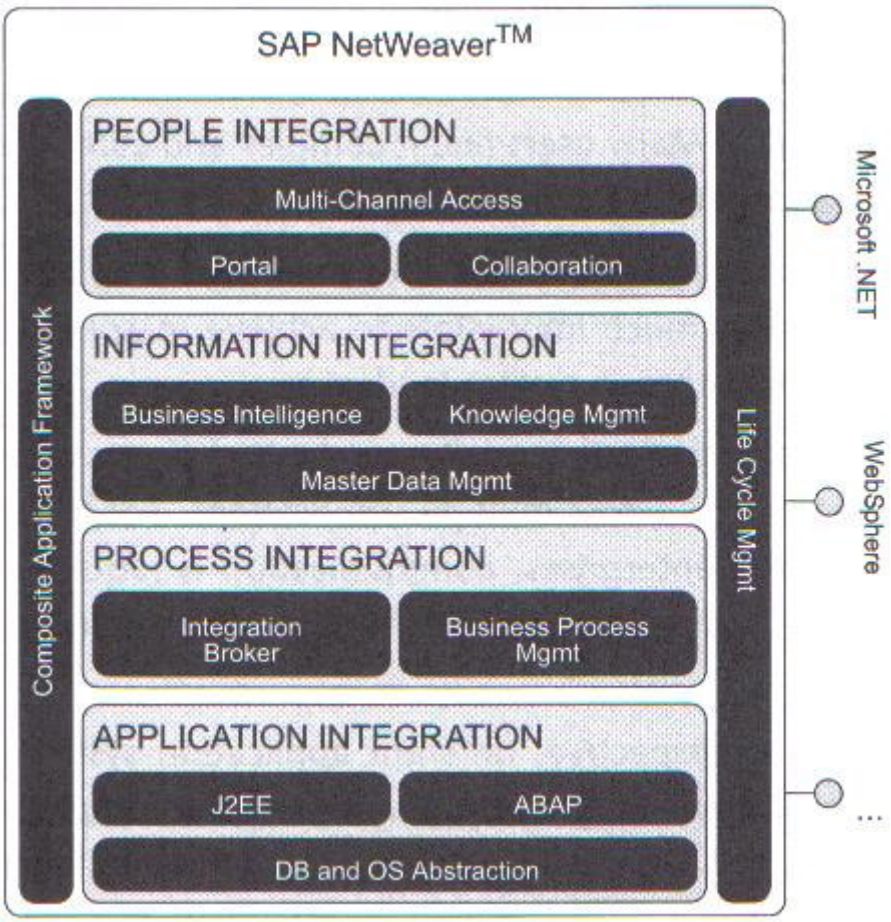

Figura 3.14 – Overview do SAP *NetWeaver*.

O SAP *NetWeaver* busca dados de aplicações ou componentes e disponibiliza serviços para as *composite applications* [SAP, 2007].

A SAP possui o SAP *NetWeaver Developer Studio* como ferramenta de desenvolvimento, que está baseada no *framework Eclipse Open-Source*.

O SAP XI (de *eXchange Infraestructure*) é o centro da arquitetura ESA. Concentra as funções de acesso a dados e aplicações através de diversos adaptadores, gerenciamento dos serviços do ambiente, monitoramento do status dos serviços e, por fim, gerenciamento dos fluxos de trabalho, fazendo o papel de uma ferramenta de *Business Process Management*

(BPM). Sua implementação faz uso de XML e múltiplos adaptadores para conectar diferentes sistemas.

3.4 Conclusão

Com base nos estudos apresentados, um comparativo se faz necessário para melhor identificar os pontos que este trabalho irá focar. Alguns critérios foram definidos para a comparação, e serão explicados a seguir:

- Acesso não intrusivo: esse critério avalia a capacidade da ferramenta de não requerer que o sistema que irá expor suas funcionalidades como serviços sofra alterações no código fonte, ao contrário do acesso intrusivo, que requer alterações no sistema que irá expor suas funcionalidades como serviços;
- Geração de *Web Services*: esse critério avalia a capacidade da ferramenta de gerar *Web Services* a partir de fontes de dados sem a necessidade de codificações;
- Catálogo de serviços: é o repositório que centraliza dados dos serviços disponíveis em um ambiente facilitando seu uso;
- Uso de adaptadores: são tecnologias que quando utilizadas adicionam algum mecanismo para facilitar a comunicação entre diferentes aplicativos;

A tabela 3.1 apresenta estes critérios comparativos entre as ferramentas utilizadas no estudo.

Tabela 3.1 – Tabela comparativa.

	Smart	*Encapsulamento de código legado*	*IBM WebSphere*	*SAP NetWeaver*	*Microsoft Biztalk Server*	*Oracle SOA Suite*
Acesso não intrusivo	Não	Não	Não	Não	Não	Não
Geração de *Web Services*	Não	Sim	Sim	Sim	Sim	Sim
Catálogo de serviços	Não	UDDI	UDDI	UDDI ou diretório próprio	UDDI ou diretório próprio	UDDI
Uso de adaptadores	Não	Não	Sim	Sim	Sim	Sim

Embora estas ferramentas ofereçam algum suporte para geração, disponibilização e configuração dos *Web Services*, um esforço para entender como trabalhar com as ferramentas, suas restrições e compatibilidades, bem como suas limitações deverá ser considerado no momento da utilização.

Além das ferramentas, os dois trabalhos acadêmicos apresentados focam em técnicas de como identificar quais requisitos de negócios dos sistemas legados serão expostos como serviços, ou, em recompor e alterar o sistema legado para gerar os *Web Services*.

Com base nos dados abordados na tabela 3.1, nota-se que o “Acesso não intrusivo” não é praticado por nenhuma das ferramentas e trabalhos do estudo realizado, sendo este critério o principal diferencial deste trabalho.

A partir dos estudos realizados, elaborou-se a proposta da arquitetura, as funcionalidades e a implementação que permitem geração de serviços a partir de sistemas legados de forma não intrusiva. Essa solução faz uso da tecnologia de *Web Services* e XML apresentados no capítulo anterior.

4 ARUBA: Uma Arquitetura para Gerações de Serviços a partir de Sistemas Legados de forma Não Intrusiva

Nos capítulos anteriores foram apresentados aspectos relacionados à tecnologia de *Web Services* e SOA, os quais dispõem de mecanismo de invocação, acesso, disponibilização e reuso de serviços. Desta forma, essas tecnologias definem uma estrutura para promover a integração de aplicações e o reuso na forma de serviços.

Este capítulo apresenta a proposta de uma arquitetura para promover o reuso de funções existentes em sistemas legados para que possam ser utilizadas em SOA sem que seja necessária nova codificação nestas aplicações. A solução utiliza adaptadores baseados em *Web Services*, denominados *drivers*, e um repositório de serviços que permite uma eficiente descrição do conteúdo das regras de negócios mapeadas, através de suas tecnologias padrões como a WSDL, SOAP e UDDI (todas baseadas em XML).

Trata-se do desenvolvimento de uma arquitetura que através de uma interface permite o mapeamento das regras de negócios para sua exposição como serviços. Assim, integrando essas tecnologias aos conceitos de sistemas legados não desenvolvidos sob a forma de *Web Services*, é possível reutilizar este código sem que haja necessidade de alterá-lo, mapeando as funcionalidades e gerar os serviços como *Web Services* para reuso em outra tecnologia, tal como, SOA.

4.1 Arquitetura de Geração de Serviços Tradicional

A construção de sistemas baseados no conceito de serviços, como já mencionado anteriormente, estabelece uma abstração entre os processos de negócios e as aplicações existentes nas organizações, porém, constitui uma atividade que requer alguns esforços e que pode ser realizada de diversas maneiras. Muitos são os padrões de tecnologia que podem ser utilizados para a geração desses serviços, cujo principal objetivo é contribuir para a integração de diferentes aplicações.

Ferramentas de integração surgiram e estão no mercado e todas requerem adaptações e alterações no código de sistemas legados para promoverem a integração para geração de *Web Services* que serão disponibilizados como serviços. Além disso, são ferramentas desenvolvidas conforme os padrões de seus fabricantes para atenderem demandas bem

restritas a arquitetura proposta. A figura 4.1 apresenta uma visão de como estas arquiteturas trabalham na integração de sistemas legados para a geração de *Web Services*. A arquitetura de integração, que constitui geralmente uma ferramenta, efetua a leitura dos arquivos do sistema legado, após isso, se faz necessário criar um *Web Services* baseado na regra de negócio que se deseja expor como serviço, e, por fim, o *Web Services* é publicado para utilização.

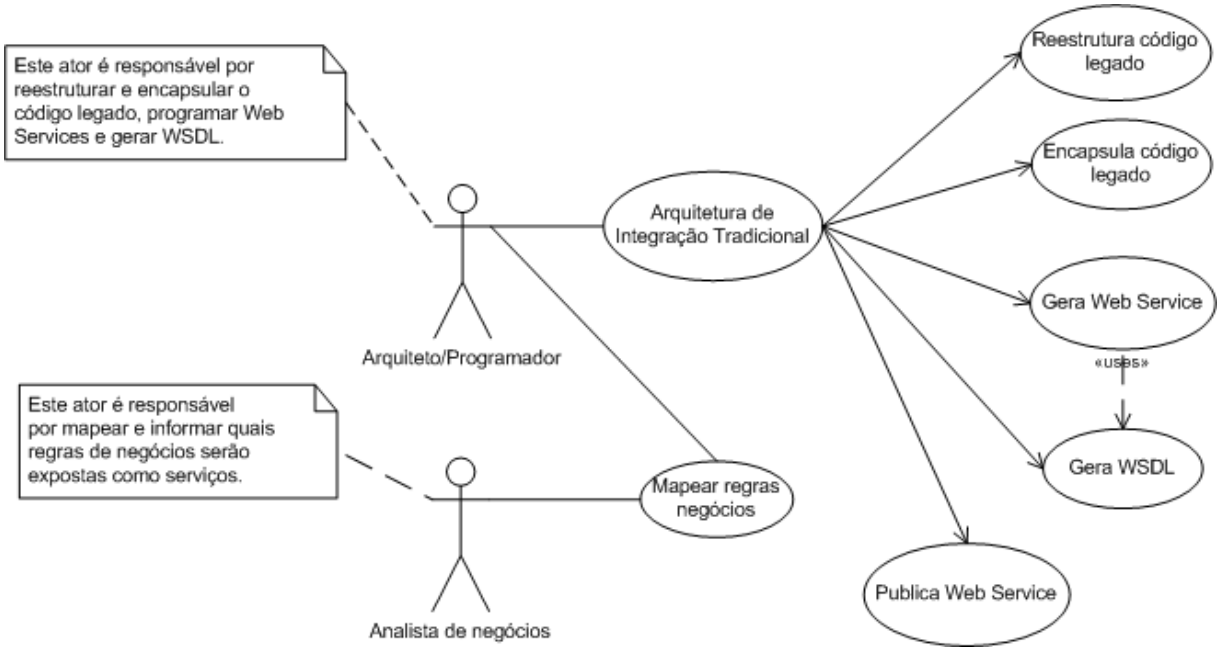

Figura 4.1 – Modelo de casos de uso: arquitetura de integração tradicional.

Uma das preocupações da arquitetura Aruba é evitar que sistemas legados estáveis necessitem ser alterados para que possam ser utilizados na SOA. Buscando otimizar este cenário, a figura 4.2 apresenta um viés onde não há necessidade de alteração do código legado, evitando assim que alterações possam comprometer o seu funcionamento.

Na figura 4.2, um mapeamento do código legado é efetuado primeiramente, após isso, ocorre a geração do *Web Services* por um método da arquitetura, assim como a geração de alguns arquivos de configuração. O modelo da arquitetura Aruba está descrito na próxima seção.

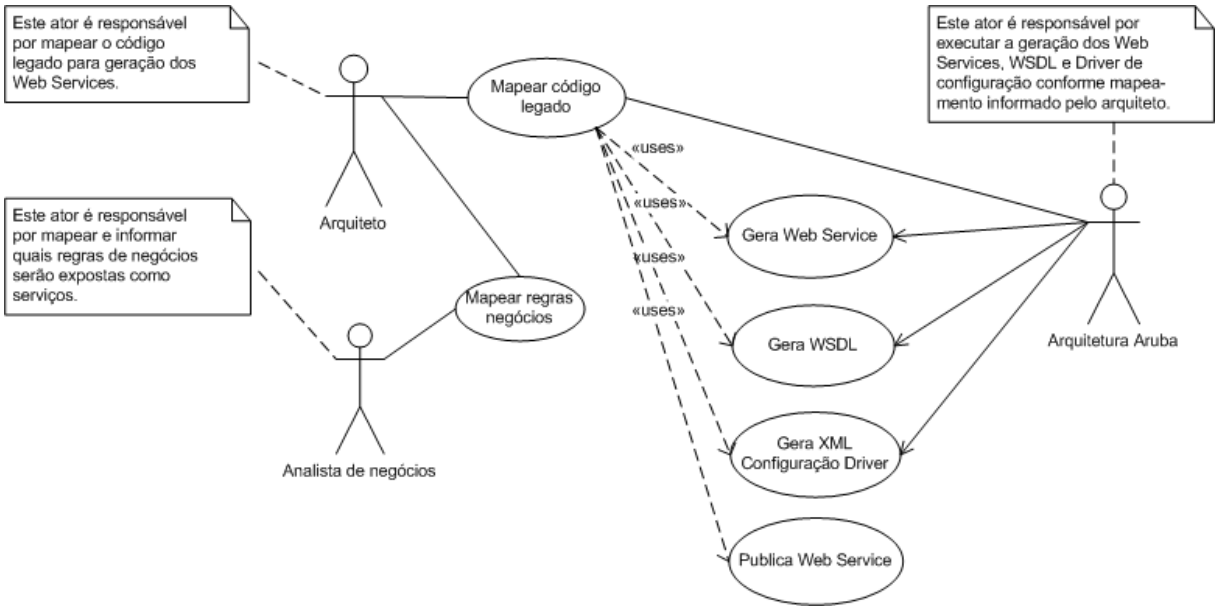

Figura 4.2 – Modelo de casos de uso: arquitetura de integração não intrusiva.

A utilização de um mecanismo facilitador, para geração de serviços, pode trazer diversas vantagens:

- Reuso do código legado;
- Gerenciamento de configuração dos serviços;
- Gerenciamento de fluxo de dados;
- Controle das regras de negócios expostas como serviços;

Porém, para que este novo cenário seja possível, são necessários cuidados em relação à forma como os dados dos sistemas legados serão decompostos em serviços, para que a lógica de negócios não seja comprometida. A proposta da arquitetura Aruba tem como característica principal ser não intrusiva, ou seja, não adicionar código no sistema legado, mas sim, expor as funcionalidades deste sistema programando uma interface com os recursos existentes de *Web Services*.

4.2 Visão Geral da Arquitetura

A figura 4.3 apresenta uma visão geral da arquitetura Aruba, onde um sistema legado "A" pode ser acessado sem a necessidade de alteração em código, tendo suas regras de negócios compostas em *Web Services,* gerando um catálogo de serviços deste sistema. Os *Web Services*

podem ser publicados num servidor Web “Servidor Web 1” e acessados por outro sistema disponibilizado em uma “*Workstation*”.

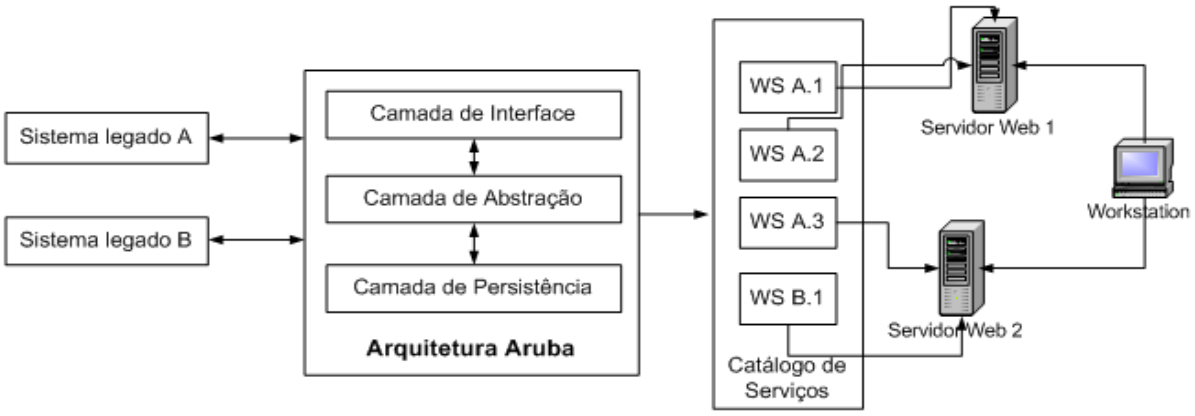

Figura 4.3 – Visão geral da arquitetura Aruba.

A arquitetura Aruba faz uso de sistemas legados. Essa utilização define um escopo desses sistemas, descrito a seguir:

- Sistemas legados devem estar desenvolvidos em Java e .Net ou possuir uma interface em que seja possível simular em *background* a interatividade do usuário com o teclado na tela do sistema;
- No caso de Java ou .Net, a compatibilidade restringe-se ao acesso a classes/métodos, não sendo possível interagir diretamente com interfaces feitas em *Java Server Pages* (JSP) ou ASPX, visto que páginas Web não podem ser mapeadas através do *Full Qualified Name* (FQN);
- No caso de Java ou .Net, cada método de uma classe deve estar codificado de forma a expor uma função atômica, sem orientação a sessão, pré-requisitos para execução ou dependência de alguma execução posterior para finalizar uma função ou método.

A arquitetura Aruba permite uma melhor utilização de funcionalidades já desenvolvidas em sistemas legados, permitindo o desenvolvimento de diversos serviços a partir das regras de negócios existentes nestes sistemas e assim possibilitando a sua melhor reutilização através da sua publicação. A arquitetura Aruba é formada por três camadas: Camada de Interface responsável por interagir com o usuário; Camada de Abstração responsável por fornecer às funcionalidades de acesso aos sistemas legados; e a Camada de

Persistência, que junto com um serviço UDDI, é responsável por manter as informações dos serviços gerados.

A arquitetura Aruba fornece uma interface de configuração possibilitando que um usuário configure um serviço que será disponibilizado num catálogo de serviços. Os serviços gerados serão baseados nas regras de negócios existentes nos sistemas legados. Diferentes serviços poderão ser configurados para compor uma regra de negócio, desde que estes serviços não dependam uns dos outros. O critério para configuração do serviço será do usuário conhecedor do sistema legado, que deverá estar de acordo com as necessidades de negócios da empresa.

A figura 4.4 apresenta uma visão mais detalhada das camadas da arquitetura Aruba. Observando a figura, percebe-se que a interface de configuração fornece um arquivo .XML que define para o *driver* o modo de acesso ao sistema legado, ou seja, as entradas, saídas e configurações necessárias ao acesso.

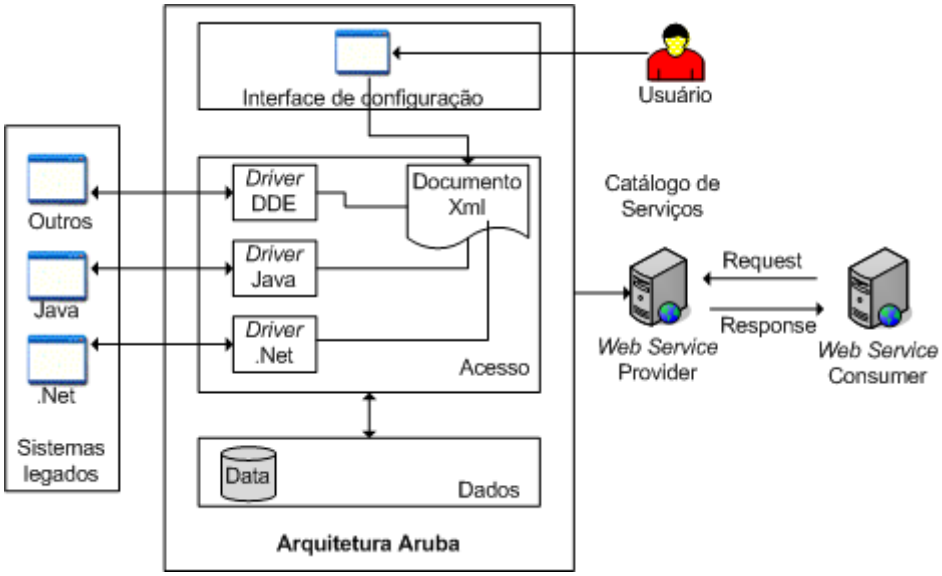

Figura 4.4 – Detalhes das camadas da arquitetura.

É importante destacar os componentes da arquitetura Aruba, que são:

- **Sistema legado**: entende-se como sistema legado qualquer sistema que não tenha sido concebido sob o conceito de SOA. Para tal, basta que ele não tenha *Web Services* ou qualquer tipo de interface para disponibilização de serviços (*sockets* ou CORBA, por exemplo), e que sigam o escopo detalhado no início da seção 4.2;

- **Dados**: compõem a base de dados da arquitetura assim como um serviço de UDDI;
- ***Drivers***: *drivers* cumprem a função de adaptadores. Eles fazem o acesso aos sistemas legados e disponibilizam esse acesso sob o conceito de serviço;
- **Catálogo de Serviços**: são os *Web Services* disponibilizados pela arquitetura sob o conceito de serviços para uso em SOA;
- **Documento XML**: documento que armazena os dados que servem como parâmetros de acesso ao sistema legado;
- **Interface de Configuração**: fornece as operações para configuração dos serviços.

As três camadas que constituem a arquitetura Aruba são:

- **Camada de interface**: responsável por fornecer suporte à configuração dos serviços que serão criados e disponibilizados pela arquitetura. Esta configuração será efetuada através do mapeamento de classes, métodos e parâmetros, e descrição da identificação dos parâmetros do serviço, como nome, sistema e descrição. Também estabelece a comunicação entre os sistemas legados e a arquitetura Aruba. A interface oferece ao usuário três tipos de *drives*: *Driver* Java, *Driver* .Net e *Driver* DDE, e permite o cadastro dos serviços que serão disponibilizados para que o *driver* efetue o acesso ao sistema legado;
- **Camada de abstração de *drivers***: responsável por fazer acesso ao sistema legado e garantir esse acesso através da geração de serviços que serão disponibilizados como *Web Services*. Como resultado da configuração realizada pelo usuário na interface de configuração, gera-se um arquivo XML de configuração para o *driver* (de acordo com a forma de acesso) e um *Web Services* que disponibilizará os parâmetros de entrada e saída configurados também pela interface de configuração;
- **Camada de persistência:** permite que os dados mapeados para a geração dos *Web Services* sejam salvos para sua utilização posterior.

4.2.1 *Drivers*

A Aruba compõe em sua arquitetura a definição de três *drivers*. Um *driver* cumpre a função de um adaptador que efetua a conexão entre o *Web Services* gerado e o sistema legado. Os três tipos de *drivers* disponibilizados pela arquitetura são apresentados nas próximas seções.

4.2.1.1 *Driver* DDE

O *driver* DDE será utilizado para acesso aos sistemas legados que permitem integração apenas através de uma interface. O *driver* simulará as entradas de um usuário diretamente na interface do sistema, como se ele, de fato, o estivesse utilizando. Além de necessitar de um console dedicado somente para ele (para simular a interação entre o usuário e o sistema legado) é certamente o mais rústico e com menor desempenho dos *drivers*.

4.2.1.2 *Driver* Java

O *driver* Java fará acesso direto as classes de negócio ou dados do sistema legado. O nome do *container*, a classe, o método, seus parâmetros e seus retornos serão configurados na interface no momento da criação de um novo serviço. Assim, quanto melhor estruturado estiver o sistema legado, melhor será o resultado final do serviço, pois possibilitará acesso a uma camada de grande abstração do sistema (seguindo o conceito de um sistema *n-tier*). Por motivos de compatibilidade, o *driver* Java será desenvolvido em Java, evitando possíveis problemas de abstração ou desempenho, comuns em métodos de integração e abstração entre linguagens.

4.2.1.3 *Driver* .Net

O *driver* .Net será idêntico ao *driver* Java, servindo para ele todas as definições feitas no item 4.2.1.2. Por motivos análogos ao *driver* Java, o *driver* .Net será desenvolvido em C#, preservando assim a compatibilidade integral entre o *driver* e o sistema legado.

4.2.1.4 Modelo do XML interpretado pelos *Drivers*

Para que seja possível a conexão entre o *Web Service* gerado e o sistema legado, a definição de um modelo de arquivo XML, comum aos *drivers*, é necessário para flexibilizar essa comunicação. A figura 4.5 apresenta o modelo do XML para armazenar a informação de acesso ao sistema legado e do serviço que será interpretado pelo *driver* correspondente. O arquivo é identificado pelas tags:

- <aplicacao> - corresponde aos dados do serviço a ser criado;

- <driver> - indica o *driver* que será utilizado;
- <nomeServico> - nome do serviço criado;
- <classe>, indica a classe que será acessada, identificada pelo FQN. Através deste parâmetro, a classe é dinamicamente encontrada dentre as classes conhecidas pela *Java Virtual Machine* (JVM), no caso do Java, ou no *Common Language Runtime* (CLR), no caso do .Net;
- <metodo> - corresponde aos dados do método a ser acessado;
- <nome> - nome do método;
- <mascara> - contém a definição do tipo de dados dos campos do método;

```
1#<?xml version="1.0">
2#<aplicacao>
3#    <driver></driver>
4#    <nomeServico></nomeServico>
5#    <classe></classe>
6#</aplicacao>
7#<metodo>
8#    <nome></nome>
9#    <mascara></mascara>
10#</metodo>
```

Figura 4.5 – Modelo do XML interpretado pelos *drivers* Java e .Net.

A figura 4.6 apresenta o modelo do XML interpretado pelo *driver* DDE. O arquivo é identificado pelas tags:

- <aplicacao> - corresponde aos dados do serviço a ser criado;
- <driver> - indica o *driver* que será utilizado;
- <nomeServico> - nome do serviço criado;
- <sequenciaInicial> - indica o caminho que deverá ser percorrido para simular a execução do sistema legado;
- <parametro> - corresponde aos dados do parâmetro a ser acessado na interface;
- <tabs> - corresponde a seqüência de tabs que serão necessários para acessar o parâmetro;
- <tipo> - contém a definição do tipo de dados do campo da interface;

```
1#<?xml version="1.0">
2#<aplicacao>
3#    <driver></driver>
4#    <nomeServico></nomeServico>
5#    <sequenciaInicial></sequenciaInicial>
6#</aplicacao>
7#<parametro>
8#    <tabs></tabs>
9#    <tipo></tipo>
10#</parametro>
```

Figura 4.6 – Modelo do XML interpretado pelo driver DDE.

4.2.1.5 Modelo do WSDL de um *driver*

O código apresentado na figura 4.7, representa o arquivo WSDL de descrição de um *driver*. Esse arquivo WSDL contém três campos principais:

- <pathXML> - identifica o caminho do arquivo XML que contêm a definição do serviço que fará uso do *driver*;
- <metodo> - identifica o método que será chamado;
- <parametro> - recebe todos os parâmetros do método do serviço que foi criado, concatenados e separados por vírgula.

```
1#<?xml version="1.0" encoding="UTF-8" ?>
2#<wsdl:definitions targetNamespace="http://server/ws/driverJava.jws"
3#xmlns:apachesoap="http://xml.apache.org/xml-soap"
4#xmlns:impl="http://server/ws/driverJava.jws"
5#xmlns:intf="http://server/ws/driverJava.jws"
6#xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
7#xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
8#xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
9#xmlns:xsd="http://www.w3.org/2001/XMLSchema">
10#<wsdl:message name="callLegacyFunctionResponse">
11# <wsdl:part name="callLegacyFunctionReturn" type="xsd:anyType" />
12#</wsdl:message>
13#<wsdl:message name="callLegacyFunctionRequest1">
14#<wsdl:part name="pathXML" type="xsd:string" />
15#<wsdl:part name="metodo" type="xsd:string" />
16#<wsdl:part name="parametro" type="xsd:string" />
17#</wsdl:message>
18#<wsdl:message name="callLegacyFunctionResponse1">
19#<wsdl:part name="callLegacyFunctionReturn" type="xsd:string" />
20#</wsdl:message>
21#<wsdl:message name="callLegacyFunctionRequest">
22#<wsdl:part name="pathXML" type="xsd:string" />
23#<wsdl:part name="method" type="xsd:string" />
24#<wsdl:part name="parameters" type="xsd:string" />
25#<wsdl:part name="returnType" type="xsd:string" /> </wsdl:message>
26#<wsdl:portType name="driverJava">
27#<wsdl:operation name="callLegacyFunction" parameterOrder="pathXML
28#method parameters returnType">
29#</wsdl:operation>
30#<wsdl:operation name="callLegacyFunction" parameterOrder="pathXML
31#metodo parametro">
32#</wsdl:operation></wsdl:portType>
33#<wsdl:binding name="driverJavaSoapBinding" type="impl:driverJava">
34#…
35#</wsdl:binding>
36#<wsdl:service name="driverJavaService">
37#<wsdl:port binding="impl:driverJavaSoapBinding" name="driverJava">
38#<wsdlsoap:address location="http://server/ws/driverJava.jws" />
39#</wsdl:port></wsdl:service></wsdl:definitions>
```

Figura 4.7 – Modelo do arquivo WSDL do *driver*.

4.2.1.6 Considerações sobre a criação de novos *drivers*

A criação de *drivers* para estender a arquitetura Aruba para uso em outros sistemas legados, que não os definidos para este trabalho é possível. Para tal, deve-se apenas observar que:

- Ainda que não obrigatório, é recomendável que o *driver* seja desenvolvido na linguagem do sistema legado a ser acessado;
- O XML de entrada deve seguir o modelo apresentado na seção 4.2.1.4. Este XML que é interpretado pelo *driver* serve como modelo genérico para outros sistemas legados, ressaltando que para o driver DDE este XML segue outro modelo mais específico;
- A forma de acesso ao sistema legado deve ser selecionada de acordo com a linguagem, não importando a arquitetura deste trabalho, utilizada para os *driver* Java e .Net.

A seguir, são descritos alguns modelos que descrevem o funcionamento da arquitetura proposta. Estes modelos são descritos a seguir usando a *Unified Modeling Language* (UML)[11].

4.2.2 Modelo de Casos de Uso

O modelo de casos de uso apresentado na figura 4.8, especifica as operações realizadas pelo usuário na interface de configuração, no contexto de utilização da arquitetura.

[11] Informações podem ser encontradas em: <http://www.uml.org>

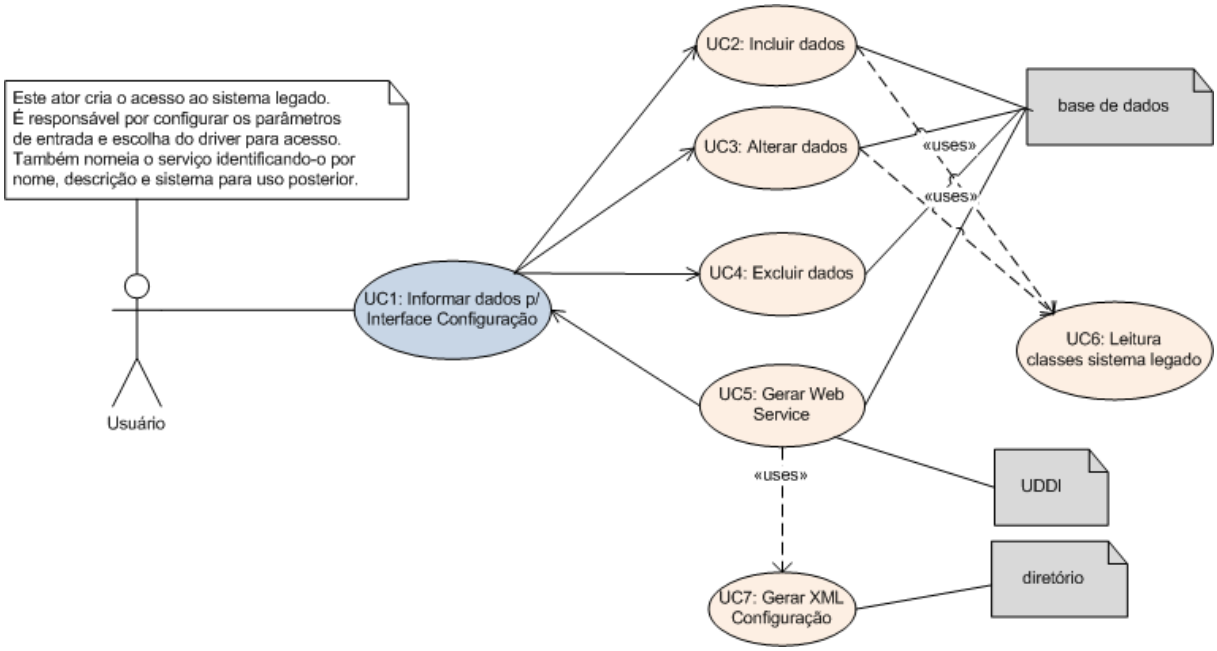

Figura 4.8 – Modelo de casos de uso: mapeamento e geração de serviços.

Segue uma descrição de cada caso de uso do modelo:

- **UC1**: Informar dados p/Interface de Configuração – o usuário informa os parâmetros necessários para a geração dos serviços. O *driver* para a qual o serviço será criado é a primeira informação a ser solicitada. Para *driver* Java e .Net, classes e métodos deverão ser mapeados, e os parâmetros, tipos de dados e parâmetros correspondentes no *Web Services* de cada um deles deverão ser informados. Já para *driver* DDE, a informação da seqüência inicial de comando deverá ser informada, assim como os tipos de dados dos campos da interface e seus correspondentes no *Web Services*. Um parâmetro "Tabs" identifica o número de separadores entre um campo e outro no acesso, informado somente para *driver* DDE. Outros dados importantes são os que compõem as informações sobre o serviço, são elas: nome do serviço, o sistema a que ele faz acesso e uma breve descrição do que faz o serviço.
- **UC2**: Incluir dados – inclui as informações na base de dados;
- **UC3**: Alterar dados – altera os dados de um serviço gerado;
- **UC4**: Excluir dados – exclui os dados de um serviço gerado;
- **UC5**: Gerar *Web Services* – após configurar as informações sobre o serviço, o *Web Services* correspondente é gerado e publicado conforme os parâmetros informados;

- **UC6**: Leitura classes sistema legado – efetua leitura das classes do sistema legado para mapear nome do método e atributos que serão adicionados no *Web Services.* Esta opção é solicitada somente para *driver* Java e .Net;
- **UC7**: Gerar XML de configuração – um arquivo, denominado de "XML de configuração" é gerado, contendo as informações referentes ao sistema legado.

As etapas que envolvem pesquisa de dados, bem como operações de inclusão, alteração e exclusão dos serviços são efetuadas através da camada de persistência da arquitetura. Ao gerar o *Web Services* correspondente ao serviço mapeado na interface de configuração, o arquivo XML de configuração também é gerado, assim, quando o *Web Services* receber uma requisição, o arquivo será consultado pelo *driver* correspondente, identificado no arquivo, e a informação será buscada no sistema legado através do método e parâmetros mapeados. O modelo de casos de uso apresentado na figura 4.9, apresenta o funcionamento interno da arquitetura em relação a camada de abstração dos *drivers*.

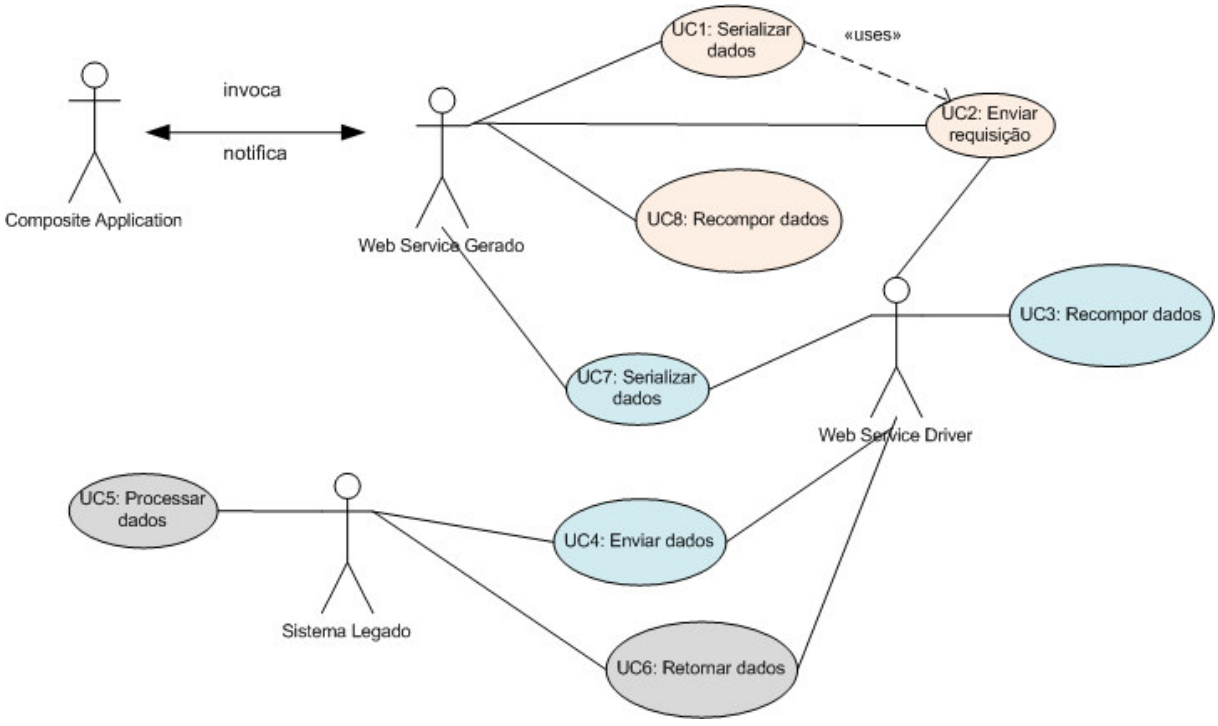

Figura 4.9 – Modelo de caso de uso: camada de abstração de *drivers*.

Segue uma descrição de cada caso de uso do modelo:

- **UC1**: Serializar dados – o *Web Services* invocado pela *composite application* recebe os dados de entrada e agrupa-os num único parâmetro;

- **UC2**: Enviar requisição – após a serialização dos dados de entrada, o *Web Services* Gerado envia uma requisição para o *Web Services Driver* correspondente, através da identificação do parâmetro <driver>, mapeado no XML de configuração;
- **UC3**: Recompor os dados – o *Web Services Driver* efetua um *cast* dos dados recebidos para objetos, de forma que obtenha os tipos de dados apropriados;
- **UC4**: Enviar dados – o *Web Services Driver* envia os dados recompostos ao sistema legado, juntamente com as informações correspondente de acesso, mapeado no XML de configuração;
- **UC5**: Processar dados – o sistema legado reage à requisição recebida e processa os dados;
- **UC6**: Retornar os dados – o sistema legado retorna o resultado da requisição para o *Web Services Driver*;
- **UC7**: Serializar dados – a função deste caso de uso é a mesma do UC1, porém o ator que realiza é o *Web Services Driver*;
- **UC8**: Recompor dados – a função deste caso de uso é a mesma do UC3, porém o ator que realiza é o *Web Services* Gerado.

O funcionamento da arquitetura está diretamente relacionado com as configurações de mapeamento dos serviços que serão disponibilizados como *Web Services*. À medida que as requisições serão efetuadas, a arquitetura através do *Web Services driver* funciona como um *middleware* entre o sistema legado, o *Web Services* Gerado e a *composite application*.

4.2.3 Modelo de Interações

O modelo apresentado na figura 4.10, apresenta o funcionamento da arquitetura de forma seqüencial, ou seja, como e quando as ações ocorrem. Este modelo refere-se às ações que o usuário executa no instante que efetua o mapeamento dos serviços.

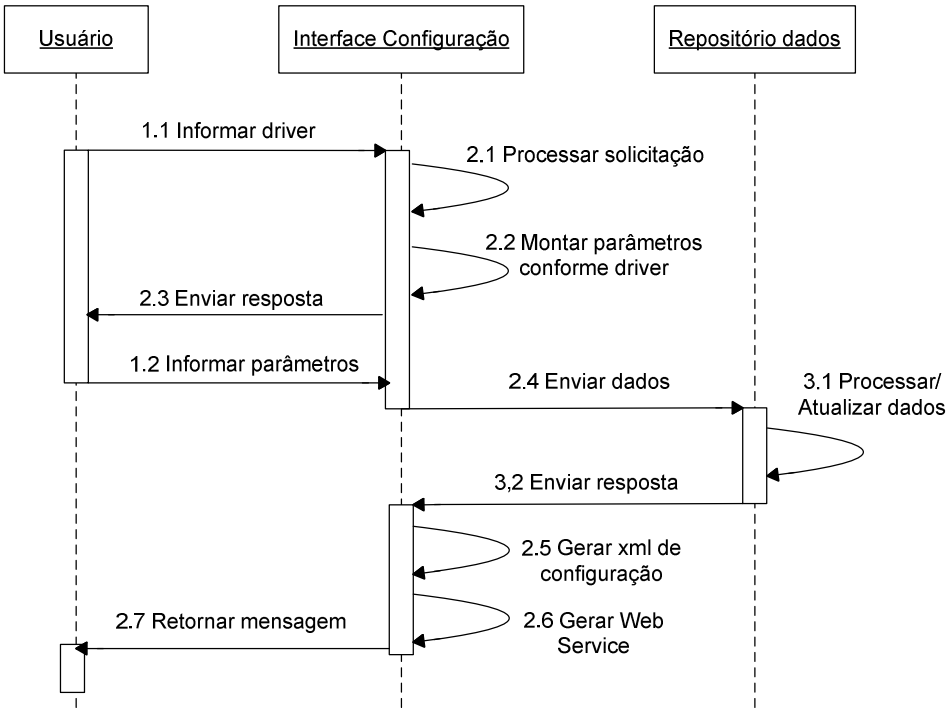

Figura 4.10 – Modelo de interações: diagrama de seqüência, mapeamento dos serviços.

Outro modelo é apresentado na figura 4.11, que está relacionado ao funcionamento do processo de abstração. Os *Web Services* gerados pela interface de configuração fazem um papel de *middleware* entre as *composite applications* e os *drivers*. O papel do serviço (*Web Services* Gerado) é apenas receber os parâmetros da *composite application*, consolidá-los em um parâmetro único, e reenviá-los ao *driver* juntamente com o identificador do XML de configuração do *driver*, que identificará a este como tratar os parâmetros e como fazer o acesso ao sistema legado.

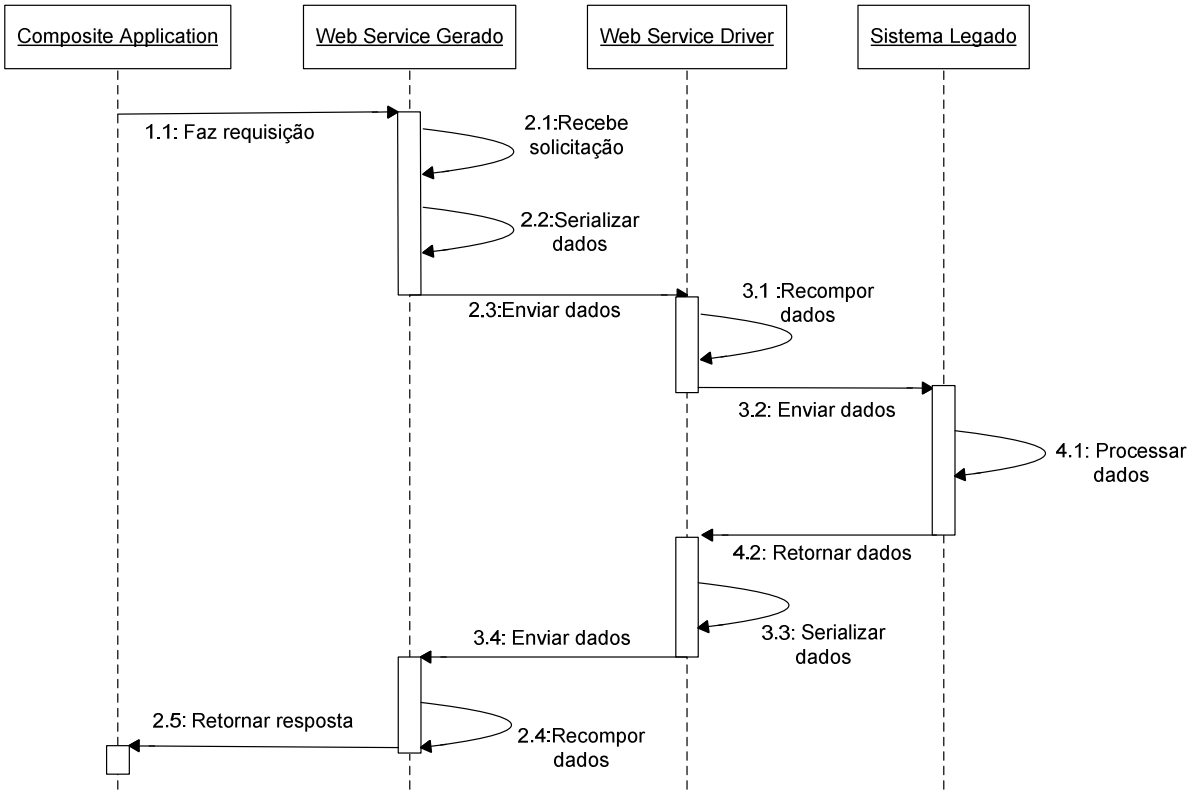

Figura 4.11 – Modelo de interações: diagrama de seqüência, abstração dos *drivers*.

A interação ocorre da seguinte forma:

1. A *composite application* faz a requisição ao serviço (*Web Services* Gerado);
2. O *Web Services* recebe a requisição com os parâmetros, serializa e envia ao *driver* (que também é um *Web Services*), juntamente com o identificador do XML que define os parâmetros de acesso ao sistema legado;
3. O *driver* recompõe os dados de entrada de forma que o sistema legado entenda e submete a ele;
4. Caso haja retorno do sistema legado, o *driver* faz sua captura e envia ao *Web Services* de serviço;
5. O retorno é enviado a *composite application*.

O serviço gerado pela arquitetura Aruba como *Web Services* possui um arquivo de descrição. O código apresentado na figura 4.12, representa o arquivo WSDL de descrição de um serviço.

```
1#<?xml version="1.0" encoding="UTF-8" ?>
2#<wsdl:definitions targetNamespace="http://server/ws/modelo.jws"
3#xmlns:apachesoap="http://xml.apache.org/xml-soap"
4#xmlns:impl="http://server/ws/andresa.jws"
5#xmlns:intf="http://server/ws/andresa.jws"
6#xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
7#xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
8#xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
9#xmlns:xsd="http://www.w3.org/2001/XMLSchema">
10#<wsdl:message name="adicionaRequest">
11#<wsdl:part name="campo1" type="xsd:int" />
12#<wsdl:part name="campo2" type="xsd:int" />
13#</wsdl:message>
14#<wsdl:message name="adicionaResponse">
15#<wsdl:part name="adicionaReturn" type="xsd:string" />
16#</wsdl:message>
17#<wsdl:portType name="modelo">
18#<wsdl:operation name="adiciona" parameterOrder="campo1 campo2">
19#<wsdl:input message="impl:adicionaRequest" name="adicionaRequest" />
20#<wsdl:output message="impl:adicionaResponse" name="adicionaResponse" />
21#</wsdl:operation>
22#</wsdl:portType>
23#<wsdl:binding name="modeloSoapBinding" type="impl:modelo">
24#<wsdlsoap:binding style="rpc"
25#transport="http://schemas.xmlsoap.org/soap/http" />
26#<wsdl:operation name="adiciona">
27#<wsdlsoap:operation soapAction="" />
28#<wsdl:input name="adicionaRequest">
29#<wsdlsoap:body
30#encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
31#namespace="http://DefaultNamespace" use="encoded" />
32#</wsdl:input>
33#<wsdl:output name="adicionaResponse">
34#<wsdlsoap:body
35#encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
36#namespace="http://server/ws/andresa.jws" use="encoded" />
37#</wsdl:output>
38#</wsdl:operation>
39#<wsdl:service name="modeloService">
40#<wsdl:port binding="impl:modeloSoapBinding" name="modelo">
41#<wsdlsoap:address location="http://server/ws/modelo.jws" />
42#</wsdl:port>
43#</wsdl:service>
44#</wsdl:definitions>
```

Figura 4.12 – Modelo do WSDL de um serviço.

O código apresentado na figura 4.13, representa um exemplo do arquivo gerado para publicação em um serviço de diretório UDDI, o UDDI *businessService* e *bindingTemplate*. Este arquivo é gerado baseado nas configurações do arquivo WSDL do serviço, apresentado

no modelo da figura 4.12. No exemplo da figura 4.13, a entidade WSDL *service* é mapeada para o *businessService* e o WSDL *port* mapeada para o *bindingTemplate*. O *businessService* contém um *categoryBag* (figura 4.13, linha 24) que indica que este serviço representa um WSDL *service*, e especifica o WSDL *namespace* e o WSDL *service name*. O *bindingTemplate* especifica o *endpoint* do serviço, e contém um conjunto de *tModelInstanceDetails*.

```
1#<businessService serviceKey="." businessKey=".">
2#<name>Modelo Service </name>
3#<bindingTemplates>
4#<bindingTemplate bindingKey="." serviceKey=".">
5#<accessPoint URLType="http">http://server/ws/modelo.jws
6#</accessPoint>
7#<tModelInstanceDetails>
8#  <tModelInstanceInfo tModelKey=".">
9#   <description xml:lang="en">
10#    O wsdl:binding
11#  </description>
12#  <instanceDetails>
13#  <instanceParms>modelo</instanceParms>
14#  </instanceDetails>
15# </tModelInstanceInfo>
16# <tModelInstanceInfo tModelKey=".">
17#  <description xml:lang="en">
18#    O wsdl:portType
19#  </description>
20# </tModelInstanceInfo>
21#</tModelInstanceDetails>
22#</bindingTemplate>
23#</bindingTemplates>
24#<categoryBag>
25#<keyedReference tModelKey="." keyName="WSDL type"
26#   keyValue="service" />
27#   <keyedReference tModelKey="." keyName="service namespace"
28#   keyValue="http://server/ws/modelo.jws" />
29#<keyedReference tModelKey="." keyName="service local name"
30#   keyValue="modeloService" />
31#</categoryBag>
32#</businessService>
```

Figura 4.13 – Modelo do UDDI *businessService* e *bindingTemplate*.

A figura 4.14 apresenta um exemplo do arquivo de mapeamento da entidade WSDL *portType* para o *tModel*. O parâmetro da tag *name* (figura 4.14, linha 3) é o mesmo do WSDL *portType*. A tag *categoryBag* (figura 4.15, linha 8) especifica o WSDL *namespace*, e indica

que o *tModel* é do tipo "*portType*". A tag *overviewDoc* (figura 4.14, linha 4) fornece um ponteiro para o documento WDSL.

```
1#<?xml version="1.0" encoding="UTF-8" ?>
2#<tModel tModelKey="">
3#<name>modelo</name>
4#<overviewDoc>
5#<overviewURL>http://server/ws/modelo.jws?wsdl
6#</overviewURL>
7#</overviewDoc>
8#<categoryBag>
9#<keyedReference tModelKey="."
10#   keyName="portType namespace"
11#   keyValue="http://server/ws/modelo.jws"/>
12#<keyedReference tModelKey="."
13#   keyName="WSDL type"
14#   keyValue="portType" />
15#</categoryBag>
16#</tModel>
```

Figura 4.14 – Modelo do UDDI *portType tModel*.

4.2.4 Modelo de Dados

O modelo de dados da arquitetura Aruba é apresentado na figura 4.15. Este modelo, junto com um serviço UDDI, é responsável por armazenar todas as informações que serão fornecidas para a geração dos serviços.

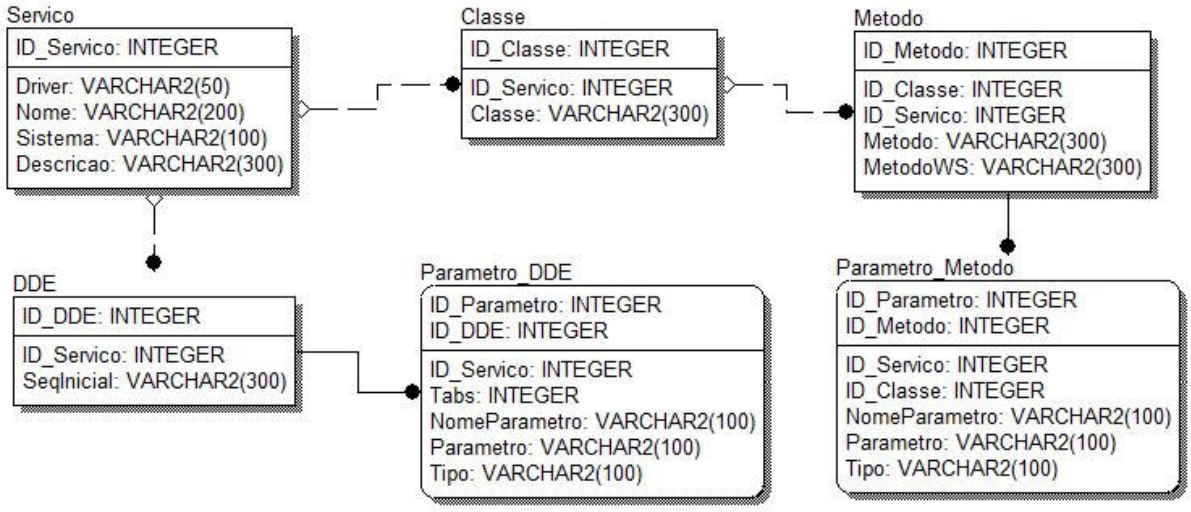

Figura 4.15 – Modelo de dados da arquitetura Aruba.

No modelo representado pela figura 4.15, encontra-se as seguintes informações de dados da arquitetura Aruba:

- **Tabela Servico**: representa o modelo do serviço a ser criado, contendo as informações referentes ao *driver* a que corresponde, o nome do serviço, sistema de referência e uma descrição do que o serviço faz;
- **Tabela Classe**: define a classe que será acessada no sistema legado;
- **Tabela Metodo e Parametro_Metodo**: define quais os métodos com seus respectivos parâmetros e tipos de dados correspondentes para serem acessados;
- **Tabela DDE**: define uma seqüência inicial de comando por onde o *driver* irá iniciar o acesso;
- **Tabela Parametro_DDE**: define por *tabs* quais parâmetros e tipos de dados serão mapeados para disponibilizar a regra de negócio;

4.3 Características da arquitetura

Uma característica importante da arquitetura Aruba está em realizar o acesso ao sistema legado sem a necessidade de alteração de código. Como a solução opera diretamente sobre a camada de abstração dos *drivers*, estes foram implementados para traduzir as informações que serão enviadas e recebidas do sistema legado.

O principal produto da arquitetura é ter uma regra de negócio mapeada e disponibilizada como serviço através de *Web Services*. Seguindo este princípio, a arquitetura gera *Web Services* que podem ser reutilizados em SOA apresentando as seguintes características:

- São fracamente acoplados;
- Abstrai a regra de negócio (desde que corretamente mapeado na interface de configuração);
- São autônomos;
- São *stateless*;
- São passíveis de descoberta.

4.4 Restrições e pré-requisitos ao uso da arquitetura

As restrições e pré-requisitos a seguir são recomendadas quanto ao uso da arquitetura:

- No caso do *driver* DDE, deve haver uma máquina dedicada para ele, de forma que uma interação do usuário com a aplicação possa ser simulada;
- No caso de Java ou .Net, a classe a ser acessada deve ser conhecida pela JVM / CLR (ex. pelo *Classpath*) de forma que possa ser identificada através de seu FQN;
- No caso de Java ou .Net, os métodos das classes a serem mapeados devem ser atômicos, ou seja, não devem necessitar do uso de sessão ou ter alguma execução pré ou pós-requerida.

4.5 Conclusão

Este capítulo teve como objetivo apresentar o modelo da arquitetura Aruba, detalhando suas três camadas, seus componentes e suas funcionalidades. A arquitetura está baseada num conjunto de definições que serviram para avaliar a aplicação de seu objetivo, bem como um conjunto de convenções de tecnologias já utilizadas anteriormente em outros trabalhos relacionados conforme apresentados no capítulo anterior.

O próximo capítulo apresentará como cada uma destas camadas foram implementadas bem como as tecnologias utilizadas neste desenvolvimento.

5 ARQUITETURA ARUBA: IMPLEMENTAÇÃO

Este capítulo detalha a implementação das camadas da arquitetura Aruba apresentadas no capítulo anterior, bem como as tecnologias que foram utilizadas no seu desenvolvimento, a estrutura e componentes resultantes. Além disso, o capítulo apresenta as telas do sistema, arquivos WSDL e XML reais gerados pela arquitetura e um passo a passo apresentando o processo de geração de serviços usando a arquitetura Aruba.

5.1 Modelo de Componentes da Arquitetura

A arquitetura Aruba foi implementada utilizando a linguagem de programação *Java 2 Enterprise Edition 5*[12] (J2EE), servidor de aplicação *Apache* TomCat 6.0[13] e banco de dados Oracle 10g *Express Edition*[14]. O protótipo da interface de configuração foi implementado JSP. Além da própria linguagem Java foram utilizadas ferramentas disponíveis na Internet, de uso gratuito, compatíveis com Java, que fazem parte dos pacotes da *Apache Software Foundation*, por exemplo, *Axis*[15] e *Xerces*[16] e o *NetBeans* 5.5[17] como ferramenta de desenvolvimento. Devido ao suporte da arquitetura a aplicações .Net, a implementação do *driver* .Net faz uso da linguagem de programação C# e servidor de aplicação *Web Microsoft Internet Information Services* (IIS) 6.0[18] e o Microsoft *Visual Studio* 2005[19] como ferramenta de desenvolvimento.

A plataforma Java foi escolhida pela sua facilidade em lidar com *Web Services*, pela ampla utilização para o desenvolvimento Web e por ser multiplataforma.

---

[12] O site oficial da Sun Java pode ser acessado em: <http://java.sun.com>

[13] O site oficial da Apache Tomcat pode ser acessado em: <http://tomcat.apache.org>

[14] O site oficial da Oracle pode ser acessado em: <http://www.oracle.com>

[15] O site oficial da Apache Axis pode ser acessado em: <http://ws.apache.org/axis/>

[16] O site oficial da Apache Xerces pode ser acessado em: <http://xerces.apache.org/>

[17] O site oficial do NetBeans em: < http://www.netbeans.org>

[18] Informações podem ser encontradas em: <http://www.microsoft.com>

[19] Informações podem ser encontradas em: <http://msdn.microsoft.com>

O *Web Server* utilizado na arquitetura foi o Apache[20], por ser uma plataforma open-source, de fácil instalação e utilização, responsável por atender as requisições do cliente via HTTP. Além disso, fornece suporte ao uso de *Web Services*.

O *Xerces* foi utilizado por ser uma biblioteca que dá suporte ao *Apache TomCat* a utilização do protocolo SOAP. Já o *Axis* foi utilizado para desenvolvimento da camada de *Web Services. Trata-se de* um *framework* desenvolvido pela Apache para construir *software* que possa processar a troca de mensagens SOAP (cliente, servidores, *gateways*, etc.), provendo mecanismo para a disponibilização e consumo dos *Web Services* em um servidor de aplicações. Dentre suas características, as principais são:

- **Flexibilidade**: a arquitetura dá liberdade para inserir extensões na *engine* para processar cabeçalhos personalizados, gestão de sistemas, entre outros;
- **Estabilidade**: define um conjunto de interfaces publicadas na qual mudam lentamente comparadas com outras da *Axis*;
- ***Deploy* orientado a componentes**: definição de redes de *handlers* reutilizáveis para implementar padrões comuns de processamento para aplicações ou distribuir para parceiros;
- ***Framework* de transporte:** abstração clara e simples para projetar transportes, sendo o núcleo da *engine* independente de transporte;
- **Tecnologia *open-source***.

Outros componentes foram utilizados na implementação das classes por fornecerem suporte na leitura e tratamento de arquivos XML e também na leitura dinâmica de classes, descritos a seguir:

- *Reflection*[21] - é comumente usado por programas que exigem a capacidade de analisar ou modificar o comportamento de aplicações em tempo de execução;
- *Document Object Model*[22] (DOM) – é uma plataforma de linguagem e interface neutra que permite programas e *scripts* acessarem e atualizarem dinamicamente conteúdos,

[20] O site oficial da Apache pode ser acessado em: <http://www.apache.org/>

[21] Informações podem ser obtidas em: <http://java.sun.com/docs/books/tutorial/reflect/index.html>

[22] O site oficial do DOM pode ser acessado em: <http://www.w3.org/DOM/>

estruturas e estilos de documentos. O documento pode ainda ser processado e os resultados dessa transformação podem ser incorporados novamente a página apresentada;

Estes componentes estão disponíveis para linguagens, assim como as utilizadas no desenvolvimento desta arquitetura. Assim, os recursos foram utilizados de forma a otimizar sua implementação.

5.2 Modelo de Classes

O modelo apresentado na figura 5.1 representa a estrutura das principais classes da arquitetura Aruba.

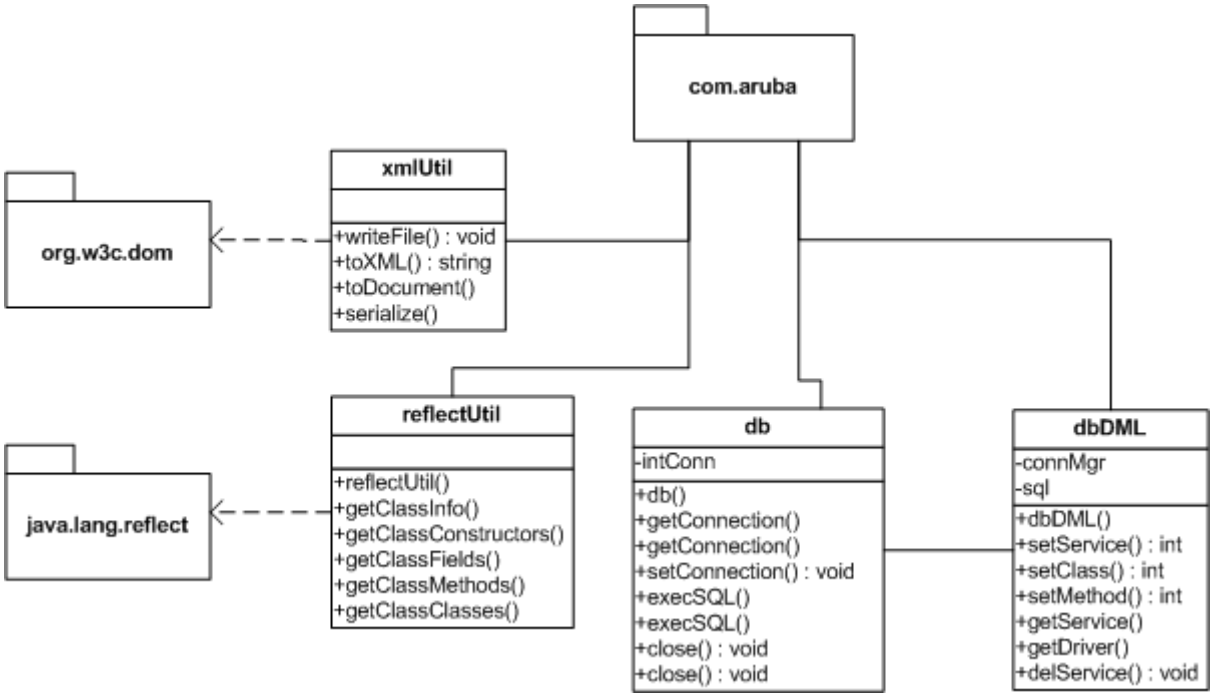

Figura 5.1 – Modelo de classes da arquitetura.

A classe db é responsável por executar as operações de conexão no banco de dados. A classe dbDML faz uso da classe db e é responsável pelas operações de manipulação de dados. A classe reflectUtil é responsável por montar uma estrutura de métodos que através do uso de *reflection* permite inspecionar as propriedades e a estrutura de uma classe e a classe xmlUtil executa operações referentes ao tratamento dos arquivos XML.

A tabela 5.1 apresenta o detalhamento dos métodos da classe db.

Tabela 5.1 – Descrição dos métodos da classe db.

Método	**Função**
getConnection	efetua a conexão com o banco de dados e retorna um objeto do tipo *connection*
setConnection	configura a conexão com o banco de dados para um objeto de classe
execSql	executa uma query no banco de dados, enviada por parâmetro
Close	fecha a conexão com o banco de dados

A tabela 5.2 apresenta o detalhamento dos métodos da classe dbDML.

Tabela 5.2 – Descrição dos métodos da classe dbDML.

Método	**Função**
setService	efetua a inclusão ou alteração dos dados de um serviço
setClass	efetua a inclusão dos dados referentes a classes ou seqüência conforme o *driver* selecionado. Caso Java ou .Net, na tabela classe, e DDE na tabela DDE
setMethod	efetua a inclusão dos dados referentes aos métodos e seus parâmetros ou campos e seus tipos de dados, conforme o *driver* selecionado
getService	obtém os dados de um serviço
getDriver	obtém os dados referentes a configuração de um serviço conforme o driver selecionado
delService	exclui os dados referentes a um serviço cadastrado

A tabela 5.3 apresenta o detalhamento dos métodos da classe reflectUtil.

Tabela 5.3 – Descrição dos métodos da classe reflectUtil.

Método	**Função**
getInfoClass	obtém as informações de uma classe, retorna um objeto *Hashtable*
getClassConstructors	obtém os construtores de uma classe, retorna um objeto *Hashtable*
getClassFields	obtém os campos de uma classe, retorna um objeto *Hashtable*
getClassMethods	obtém os métodos de uma classe, retorna um objeto *Hashtable*
getClassClasses	obtém as dependências entre classes, retorna um objeto *Hashtable*

A tabela 5.4 apresenta o detalhamento dos métodos da classe xmlUtil.

Tabela 5.4 – Descrição dos métodos da classe xmlUtil

Método	**Função**
writeFile	efetua a geração de um arquivo xml a partir de uma string
toXML	transforma o resultado de um *resultSet* em XML
toDocument	cria um objeto *Document* a partir de um *resultSet* com elementos definidos para o arquivo
Serialize	obtém os dados de um serviço

5.3 Drivers

Os *drivers* funcionam como *Web Services* para o *driver* Java e .Net e como uma classe para o DDE. Um trecho de código do *driverJava* é apresentado na figura 5.2. O *Web Services driverJava.jws* contém o método que é invocado por um *consumer*. O método *callLegacyFunction* (figura 5.2, linha 1) recebe como parâmetro uma *string* contendo o *pathXML*, uma *string* representando o método e outra *string* representando os parâmetros. O parâmetro *pathXML* identifica o caminho do arquivo XML de configuração que será interpretado pelo *driver*. O parâmetro *metodo* identifica o nome do método que será lido no arquivo e o parâmetro *parametro* identifica os parâmetros do método que será invocado.

O método callLegacyFunction interpreta o arquivo XML de forma a obter as informações necessárias para realizar o acesso ao sistema legado. As informações contidas no arquivo XML são lidas a partir do comando *Document doc = docBuilder.parse (new File(pathXML))*, (figura 5.2, linha 8).

```
1# public String callLegacyFunction(String pathXML, String metodo, String
parametro)
2# //Variáveis de buffer
3# String classe = "";
4# String mascara = "";
5# //Recupera informações do XML
6# DocumentBuilderFactory docBuilderFactory =
DocumentBuilderFactory.newInstance();
7# DocumentBuilder docBuilder = docBuilderFactory.newDocumentBuilder();
8# Document doc = docBuilder.parse (new File(pathXML));
9# //Recupera classe
10# NodeList listOfRows = doc.getElementsByTagName("aplicacao");
11# Node rowNode = listOfRows.item(0);
12# Element rowElement = (Element)rowNode;
13# NodeList aplicacaoList = rowElement.getElementsByTagName("classe");
14# Element aplicacaoElement = (Element)aplicacaoList.item(0);
15# NodeList textAplicacaoList = aplicacaoElement.getChildNodes();
16# classe = ((Node)textAplicacaoList.item(0)).getNodeValue().trim();
17# //Procura nos metodos da classe qual deve ser usado
18# listOfRows = doc.getElementsByTagName("metodo");
19# int totalRows = listOfRows.getLength();
20# for(int s=0; s<listOfRows.getLength() ; s++){
21# rowNode = listOfRows.item(s);
22# if(rowNode.getNodeType() == Node.ELEMENT_NODE){
23# rowElement = (Element)rowNode;
24# //Recupera método
25# NodeList MetodoList = rowElement.getElementsByTagName("nome");
26# Element MetodoElement = (Element)MetodoList.item(0);
27# NodeList textMetodoList = MetodoElement.getChildNodes();
28# //Verifica se este é o método que deve ser usado
29# if
(((Node)textMetodoList.item(0)).getNodeValue().trim().equals(metodo)){
30# NodeList MascaraList = rowElement.getElementsByTagName("mascara");
31# Element MascaraElement = (Element)MascaraList.item(0);
32# NodeList textMascaraList = MascaraElement.getChildNodes();
33# mascara = ((Node)textMascaraList.item(0)).getNodeValue().trim();
34# }//IF
35# }//IF
36# }//for
```

Figura 5.2 – Trecho do código do *driverJava* que interpreta o arquivo XML.

As informações contidas no arquivo XML são lidas identificando-se os nodos <aplicacao>,<servico>, <classe>, <metodo>, <mascara> identificados no código (figura 5.2, linhas 10 a 30).

Após interpretar o arquivo XML, é necessário organizar os parâmetros recebidos na chamada do método com as respectivas máscaras. O trecho de código da figura 5.3 apresenta a implementação do acesso a classe legada. Este é efetuado via *reflection* através da chamada

Class cl = Class.forName(classe), (figura 5.3, linha 15), passando a classe a ser invocada por parâmetro.

```
1# //Organiza os parâmetros
2# String[] arrParametrosStr = parametro.split(",");
3# String[] arrMascara = mascara.split(",");
4# Object[] arrParametros;
5$ arrParametros = new Object[arrMascara.length];
6# for(int j=0;j<arrMascara.length;j++){
7#   if (arrMascara[j].startsWith("int")){
8#       arrParametros[j] = new Integer(arrParametrosStr[j]);
9#   }else{
10#      arrParametros[j] = arrParametrosStr[j];
11#   }
12# }
13# //Cria o objeto que permite o acesso ao sistema legado
14# //Invoca classe a ser acessada via reflection
15# Class cl = Class.forName(classe);
16# java.lang.reflect.Constructor constructor = cl.getConstructor(null);
17# Object invoker = constructor.newInstance(null);
18# Method[] arrMethod = cl.getMethods();
19# int i = 0;
20# for (Member mbr : cl.getMethods()) {
21# if (((Method)mbr).toGenericString().indexOf("."+metodo+"(")>0) break;
22#   i++;
23# }
24# //Invoca o método passando seus parametros
25# Object result = arrMethod[i].invoke(invoker, arrParametros );
26# return (String)result;
```

Figura 5.3 – Código que invoca classe e método do sistema legado.

Os métodos são acessados via código *Method[] arrMethod = cl.getMethods()*, (figura 5.3, linha 18). Cada método é analisado e uma verificação é efetuada para comparar com o método que consta no arquivo XML. Caso método corresponde ao arquivo, este é invocado através da chamada *Object result = arrMethod[i].invoke(invoker, arrParametros)*, (figura 5.3, linha 25) e o resultado é retornado através da chamada *return (String)result*, (figura 5.3, linha 26).

5.4 Interface de configuração

A interface de configuração para geração dos serviços deve oferecer ao usuário a possibilidade de gerar um serviço conforme o tipo de *driver* selecionado. A interface deve

permitir a inclusão das informações necessárias para efetuar o acesso posterior ao sistema legado, incluindo: dados da classe, métodos e parâmetros, bem como informações do sistema, como nome e descrição. Desta forma, é necessário o uso de uma base de dados para armazenamento das informações. Outra funcionalidade disponibilizada por essa interface é a consulta dos serviços gerados, permitindo o acesso às informações armazenadas no repositório local. A figura 5.4 apresenta a interface principal do protótipo implementado, onde é possível escolher uma funcionalidade tanto para criação de um novo serviço, alteração, exclusão ou consulta da configuração de um serviço já realizada.

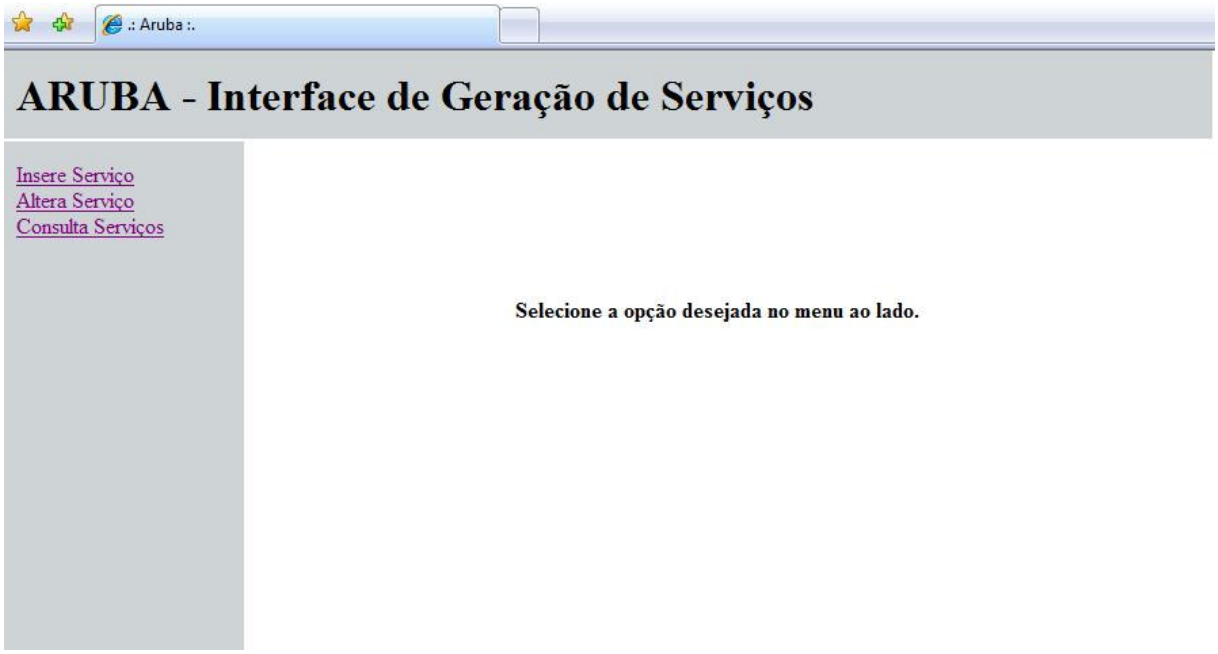

Figura 5.4 – Tela principal da Interface de Configuração.

A figura 5.5 apresenta a tela de mapeamento inicial. A primeira informação solicitada para que o usuário preencha é o "Driver", as opções disponíveis aparecem conforme apresenta a figura.

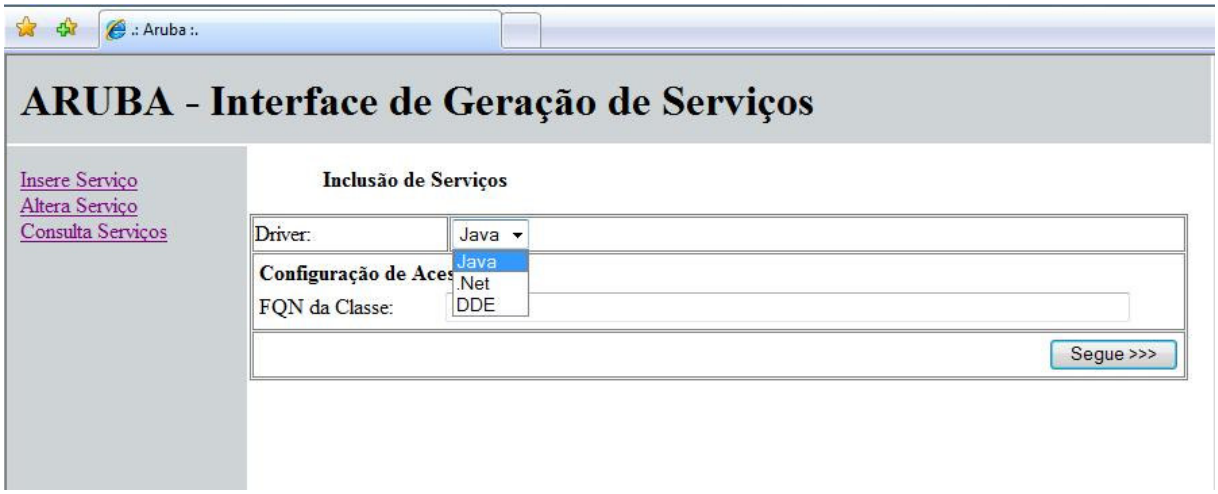

Figura 5.5 – Tela de mapeamento inicial.

Caso o *driver* selecionado seja Java ou .Net, a informação do FQN da classe deve ser preenchida no campo "FQN da Classe". O usuário informa o nome da classe, por exemplo, com.cliente.distribuicao.pedido, e pressiona o botão "Segue", conforme apresenta a figura 5.6. O sistema irá buscar dinamicamente todos os métodos que estão disponíveis pela classe, e traz as informações referentes na tela conforme apresenta a figura 5.7.

Figura 5.6 – Informação do FQN da classe (*driver* Java e .Net).

A figura 5.7 apresenta a tela de cadastro do serviço após a informação do "FQN da Classe".

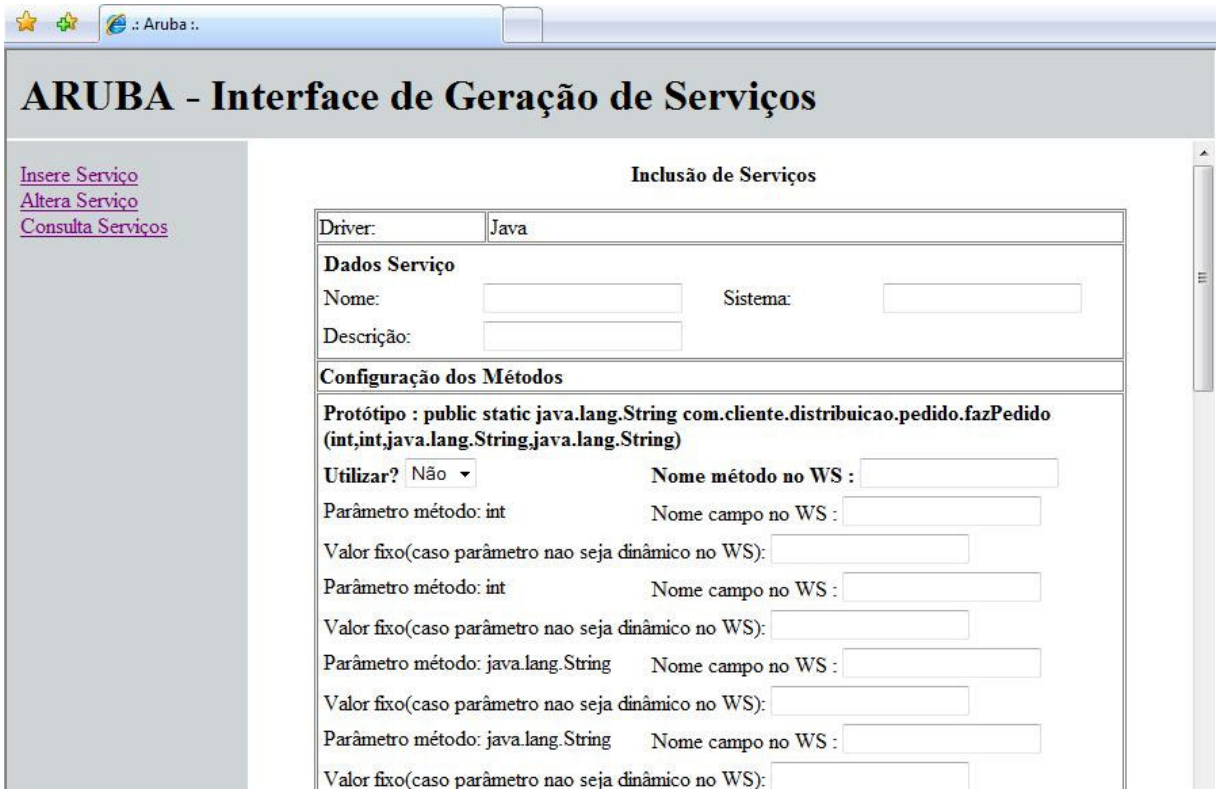

Figura 5.7 – Tela de cadastro do serviço (*driver* Java e .Net).

Na figura 5.7, as seguintes informações são apresentadas: *driver* selecionado, dados do serviço a ser criado, e para cada método que consta na classe, é montado um quadro que serve para configuração do método no *Web Services*, com as seguintes informações:

- **Protótipo**: informa os dados de declaração do método na classe, ou seja, nome, retorno e parâmetros;
- **Utilizar**: permite que o método seja incluído ou não no *Web Services*, caso sim, o nome correspondente no *Web Services* deve ser informado;
- **Parâmetro método**: informa a declaração do tipo de dado do parâmetro, e solicita o nome do campo correspondente no *Web Services*;
- **Valor fixo**: permite informar um valor fixo ao parâmetro do método, caso não necessite ser enviado pelo *consumer* na sua invocação. Este valor fica armazenado no método configurado no *Web Services* para ser utilizado, porém sem necessidade de declaração na chamada do método.

Após efetuada a configuração dos métodos que serão disponibilizados no *Web Services*, ao pressionar no botão “Gerar Serviço” será gerado o arquivo XML de configuração

necessário para posterior interpretação pelo *driver* e o *Web Services* do serviço criado assim como os arquivos XML necessários para publicação em um serviço de UDDI. Os dados informados são também armazenados no banco de dados para manutenção e consulta.

A figura 5.8 apresenta a tela de mapeamento para *driver* DDE. As informações solicitadas para configuração deste serviço difere das necessárias para o *driver* Java e .Net. Na configuração de acesso, o campo "Sequencia Inicial", corresponde ao atalho para chegar ao sistema, por exemplo, <CTRL+ESC><UP><UP><UP><ENTER>. Na configuração dos métodos, o campo "Sequência para chegar ao campo", corresponde ao atalho para o campo, por exemplo, <ALT+F>O<ENTER>, o campo "Tipo de dado", corresponde ao tipo do parâmetro e o campo "Nome Parâmetro", o nome correspondente no *Web Services*. As demais informações são referentes ao serviço a ser gerado.

Figura 5.8 – Tela de mapeamento para *driver* DDE.

A figura 5.9 apresenta a tela de consulta dos serviços gerados pela interface de configuração. Nesta tela, é possível excluir um serviço cadastrado ou obter um detalhamento dos dados do serviço selecionado, conforme apresentado na figura.

Figura 5.9 – Tela de consulta dos serviços gerados.

A figura 5.10 apresenta a tela de detalhe das informações de um serviço selecionado, onde classes, métodos e parâmetros vinculados a este serviço podem ser visualizados. Este exemplo refere-se a um serviço mapeado de uma classe Java.

Figura 5.10 – Detalhamento do serviço gerado.

5.5 Diretrizes para utilização da arquitetura

5.5.1 Análise do sistema legado

Para a utilização da arquitetura Aruba, deve-se, primeiramente, extrair do sistema legado, algumas informações. São elas:

- Linguagem de Programação. A arquitetura só dá suporte para sistemas legados desenvolvidos em Java, .Net e DDE;
- Definição das regras de negócios que estarão disponíveis como serviços, por um analista de negócios;
- Análise de um arquiteto, conhecedor da estrutura do sistema legado, para configuração dos serviços;

5.5.2 Configuração dos serviços

Com essas informações, é necessário que sejam configuradas as informações dos serviços para geração dos *Web Services* correspondentes, da seguinte forma:

- Acesso a interface de configuração dos serviços;
- Acesso a tela de inclusão de serviços para seleção do *driver*. Com o *driver* selecionado, a tela correspondente ao mapeamento é apresentada e as configurações de métodos devem ser informadas.

6 APLICAÇÃO DA ARQUITETURA ARUBA (ESTUDO DE CASO)

Este capítulo trata do desenvolvimento de um protótipo da utilização da arquitetura Aruba. A aplicação da arquitetura Aruba em estudo de caso visa possibilitar a sua avaliação em um ambiente de testes para que seu comportamento possa ser comparado à utilização de uma aplicação sem ela. Além disso, a arquitetura Aruba atua no *background* da implementação de sistemas, de modo que, somente programadores, consigam avaliar custo/tempo de desenvolvimento de *Web Services* utilizando ou não a arquitetura.

O estudo de caso apresentado neste capítulo trata de um projeto real de implementação de SOA realizado por uma empresa de *software* para uma empresa multinacional do setor químico-farmacêutico.

O capítulo apresenta a descrição do sistema utilizado, uma estrutura das classes utilizadas para mapeamento das regras de negócios que serão disponibilizadas como serviços, e de que forma a geração dos serviços é efetuada na interface de configuração de serviços, assim como uma comparação entre os resultados obtidos com e sem uso da arquitetura Aruba.

Para avaliação da arquitetura Aruba será apresentado um comparativo entre uma *Work Breakdown Structure*[23] (WBS) do projeto original do sistema com a WBS referente à mesma implementação utilizando a arquitetura Aruba. Nesta WBS, os valores referentes às etapas onde a Aruba não apresenta nenhuma contribuição, por exemplo, especificação e desenvolvimento das interfaces, serão mantidos e os resultados destas etapas no projeto original serão aproveitados.

Ao final do capítulo, é apresentado o resultado dos testes de custo/tempo, bem como uma discussão sobre os resultados obtidos.

6.1 Estudo de Caso: Sistema de Distribuição

A empresa de *software* contratada para desenvolver o projeto, objeto deste estudo de caso, desenvolve portais corporativos e soluções Web relacionadas à gestão da informação e do

[23] WBS – Representa a organização de tarefas ao longo do tempo, com seu custo e duração de execução.

conhecimento, além de prestar serviço de consultoria e manutenção evolutiva para projetos. Especialista nas tecnologias SAP *NetWeaver*, Microsoft .Net, *Sharepoint* e Java, possui 11 anos de atuação no mercado com um total de aproximadamente 200 profissionais em 5 capitais do Brasil. Através do contato com o Diretor de Tecnologia foi possível realizar o estudo de caso apresentado neste trabalho, que proporcionou a sua realização com um experimento real. Como a empresa atua em diferentes segmentos de desenvolvimento de *software*, e atualmente tem direcionado alguns de seus projetos para soluções que utilizam SOA, tornou-se possível a escolha de um desses projetos para realizar o experimento.

O projeto escolhido foi solicitado por uma empresa multinacional do setor químico-farmacêutico, que identificou a necessidade de construir um sistema que contemplasse a integração de alguns sistemas legados, que estão distribuídos entre suas sedes. As informações do sistema e projeto apresentadas nas próximas seções foram obtidos pelo Diretor de TI e repassadas para utilização neste experimento. O sistema encontra-se implantado na empresa contratante.

6.1.1 Análise do Sistema

A empresa contratante trabalha com a produção e comercialização de uma ampla gama de produtos. Para a comercialização, a empresa não efetua venda direta ao consumidor em alguns países (incluindo o Brasil), mas faz uso de distribuidores. A venda é efetuada através de sistemas de distribuição, que, por sua vez não são integrados, restringindo-se a apenas uma sede e que demandam um alto custo de manutenção. Assim, a empresa contratante optou em construir um novo sistema, voltado para SOA, de modo a reutilizar ao máximo os seus sistemas legados, padronizar todos esses sistemas em uma única arquitetura e interface e possibilitar customizações inerentes aos diferentes países de forma rápida e com o menor custo possível.

O sistema construído, denominado “Sistema de Distribuição” possui as funções a seguir:

- Registrar e manter os pedidos dos produtos de diferentes distribuidores;
- Auxiliar, através da execução de um processo externo ao sistema, na emissão das faturas dos pedidos dos distribuidores;

- Compatibilizar as unidades do estoque da empresa com as dos distribuidores, por exemplo, em alguns países usam-se litros e gramas, em outros, galões e onças. No Brasil vendem-se alguns produtos em embalagens de 20 comprimidos, em outros de 18;
- Compatibilizar os nomes dos produtos entre a empresa e os distribuidores (produtos não necessariamente têm o mesmo nome em todos os países).

O diagrama de casos de uso apresentado na figura 6.1 descreve o fluxo das informações entre o Sistema de Distribuição e o cliente.

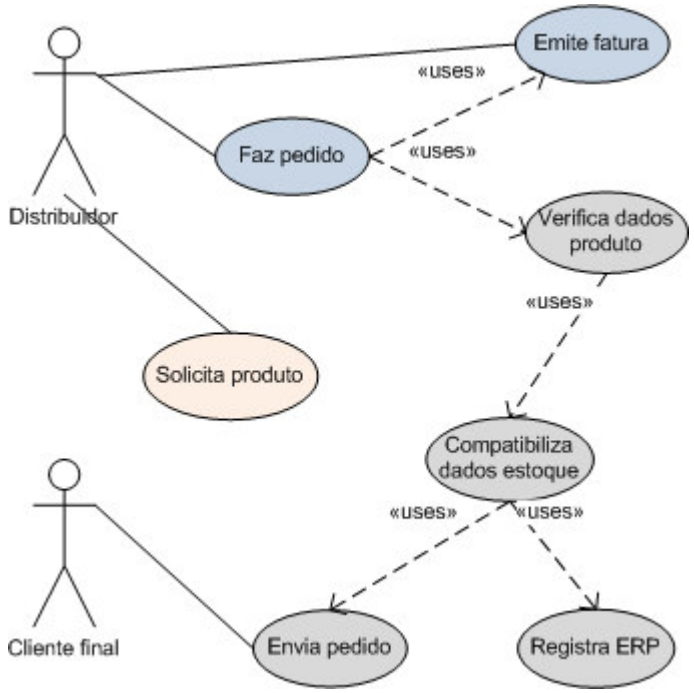

Figura 6.1 – Diagrama de casos de uso.

A arquitetura do sistema é bastante distribuída, sendo:

- O sistema de pedido dos distribuidores está implementado em Oracle *Forms* situado em outro país;
- O sistema de emissão de faturas está contemplado no sistema ERP nacional da empresa, sendo o SAP;
- O sistema de conversão de unidades não existia na empresa e foi desenvolvido no escopo do projeto original executado pela empresa contratada;

- O banco de dados, com os dados de todos os produtos, está localizado em outro país, e é dito centralizado, sendo Oracle 8.

O projeto de desenvolvimento de integração entre os diferentes sistemas envolvidos, não fez uso de uma ferramenta de desenvolvimento para geração de *Web Services*. Uma análise inicial dos diferentes sistemas envolvidos foi realizada para mapear as regras de negócios que seriam disponibilizadas como serviços e contempladas no novo sistema. Com as regras de negócios mapeadas e especificadas, a arquitetura foi projetada e o sistema construído.

O sistema foi desenvolvido com base nas tecnologias *Java 2 Enterprise Edition* (J2EE), utilizando servidor de aplicações JBoss[24] e banco de dados Oracle. No desenvolvimento do novo sistema, foram implementadas classes que faziam integração com os sistemas legados de pedido e fatura, e mapeados os métodos que seriam disponibilizados como serviço, e com base nas classes implementadas, os *Web Services* foram construídos.

6.1.2 Definição de papéis envolvidos

Para que seja possível mensurar diferenças e obter conclusões a partir do projeto desenvolvido originalmente e o projeto feito com o uso da Aruba, os perfis dos profissionais envolvidos serão listados, descritos[25] e precificados[26] na tabela 6.1.

[24] O site oficial do JBoss pode ser acessado em : < http://www.jboss.org>

[25] De acordo com os papéis existentes na grande maioria das empresas de desenvolvimento de *software*, empresas com processos "CMMi *Like"* e/ou MSF (*Microsoft Solution Framework).*

[26] Valor médio entre região sul e sudeste do profissional referente a salário acrescido de impostos e *overhead* da empresa. Valores oriundos de pesquisa realizada por uma empresa de RH em carácter privado entre Janeiro e Abril de 2007.

Tabela 6.1 – Descrição e valor/hora dos profissionais.

Função	**Breve Descrição**	**Valor/Hora**
Analista de Sistemas	Responsável por fazer o levantamento das regras funcionais e premissas do sistema.	R$ 65,92
Programador	Responsável pela codificação do sistema.	R$ 50,00
Gerente de Projeto	Responsável pela condução do projeto como um todo.	R$ 69,52
Programador HTML	Codificador da parte referente a interface do sistema.	R$ 38,90
Analista de Testes	Responsável pelo entendimento e definição dos casos de teste do sistema.	R$ 44,51
Testador	Responsável pela execução dos casos de teste definidos.	R$ 33,66
Projetista de *Software*	Responsável por garantir, junto aos programadores, o andamento do desenvolvimento assim com o uso de melhores práticas.	R$ 71,56
Arquiteto de *Software*	Responsável pela definição técnica do sistema.	R$ 81,16

6.1.3 Geração dos serviços utilizando Aruba

O Sistema de Distribuição foi desenvolvido na linguagem de programação Java, sendo, portanto, compatível com um dos *drivers* disponibilizados pela arquitetura.

O uso da arquitetura Aruba está em permitir a geração dos serviços com base nas classes implementadas no sistema, que foram criadas para integração com os sistemas legados. Por questões de avaliação da arquitetura Aruba, algumas regras de negócios foram previamente selecionadas para utilização neste estudo de caso, que são:

- Classes e métodos referentes a interface de pedidos;
- Classes e métodos referentes a interface de faturas;
- Classes e métodos referentes a interface de produtos;

- Classes e métodos referentes a interface de conversão de unidades;

O diagrama de classes apresentado na figura 6.2, descreve melhor as classes que terão seus métodos gerados como serviços e os parâmetros correspondentes.

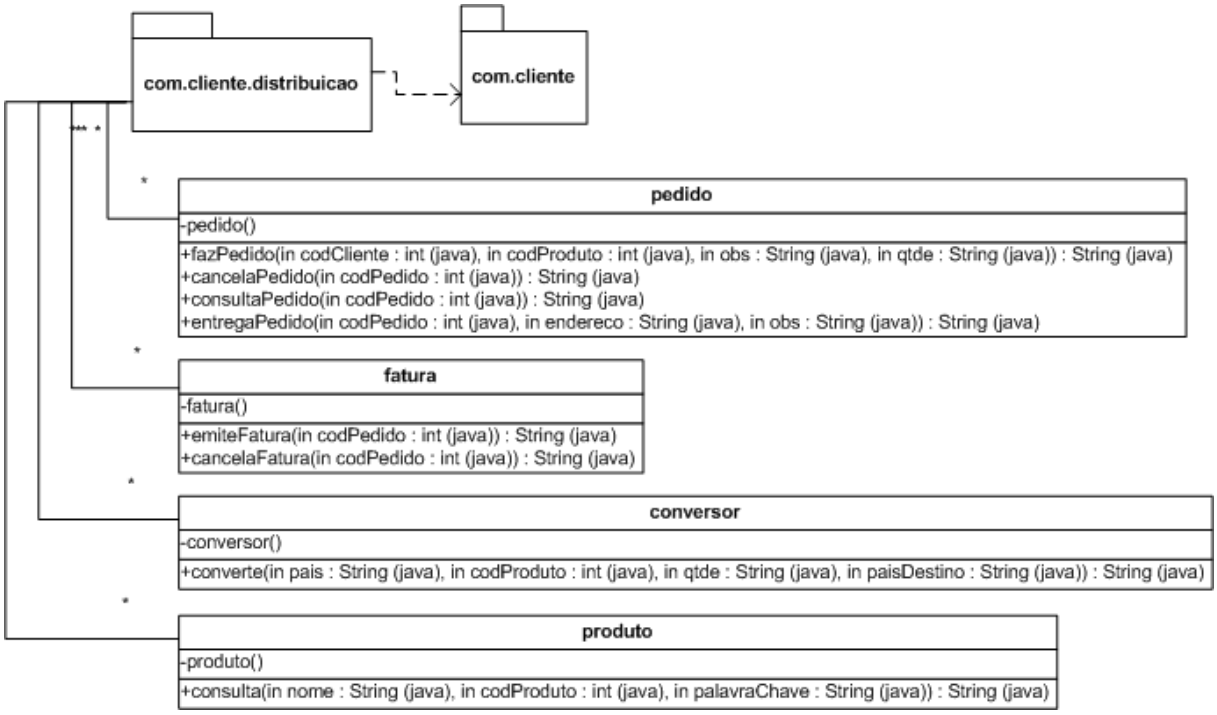

Figura 6.2 – Diagrama de classes do Sistema de Distribuição.

A classe pedido é responsável por executar as operações referentes a um pedido. Como o sistema de pedidos está desenvolvido em Oracle Forms, esta classe através de seus métodos, efetua integração com este sistema para realizar as operações referentes aos pedidos. A classe fatura é responsável pelas operações relacionadas à emissão e cancelamento de uma fatura para um determinado pedido. Para realizar esta operação, efetua integração com o sistema de emissão de faturas em SAP, contemplado num dos módulos do ERP da empresa. A classe conversor é responsável por converter a unidade de um produto de um país para outro. O conversor de unidades foi uma nova funcionalidade construída, portanto, o método utilizado não faz integração com algum outro sistema legado da empresa. A classe produto é responsável pela operação de consulta dos produtos cadastrados no banco de dados da empresa. Esta classe implementa em seu método um acesso a este banco de dados para obter as informações.

A tabela 6.2 apresenta o detalhamento dos métodos da classe pedido.

Tabela 6.2 – Descrição dos métodos da classe pedido.

Método	**Função**
fazPedido	Efetua um pedido de um produto para o cliente conforme parâmetros enviados
cancelaPedido	Efetua o cancelamento de um pedido
consultaPedido	Efetua a consulta de um pedido
entregaPedido	Confirma a entrega de um pedido conforme parâmetros enviados

A tabela 6.3 apresenta o detalhamento dos métodos da classe pedido.

Tabela 6.3 – Descrição dos métodos da classe fatura.

Método	**Função**
emiteFatura	Efetua a emissão de uma fatura para um pedido específico
cancelaFatura	Efetua o cancelamento de uma fatura para um pedido específico

A tabela 6.4 apresenta o detalhamento dos métodos da classe pedido.

Tabela 6.4 – Descrição dos métodos da classe conversor.

Método	**Função**
converte	Efetua a conversão da unidade de um produto conforme o país

A tabela 6.5 apresenta o detalhamento dos métodos da classe pedido.

Tabela 6.5 – Descrição dos métodos da classe produto.

Método	**Função**
consulta	Efetua a consulta dos produtos cadastrados no sistema

Com as classes conhecidas do sistema, o próximo passo é acessar a interface de geração de serviços da Aruba e efetuar o mapeamento dos serviços a serem gerados como *Web Services*.

A figura 6.3 apresenta a tela da interface de configuração com o *driver* Java selecionado e o FQN da classe pedido informado.

Figura 6.3 – Mapeamento do *driver* e FQN da Classe pedido.

A figura 6.4 apresenta a tela de inclusão de serviços com os métodos da classe informada disponíveis para configuração. O método *fazPedido* é apresentado juntamente com seus atributos. O nome do método e seus parâmetros podem ser alterados para geração do serviço como *Web Services*, conforme apresenta a figura 6.4. O método *fazPedido* corresponde no *Web Services* "fazerPedido " e os parâmetros são identificados por "codCliente", "codProduto", "Observação" e "Quantidade".

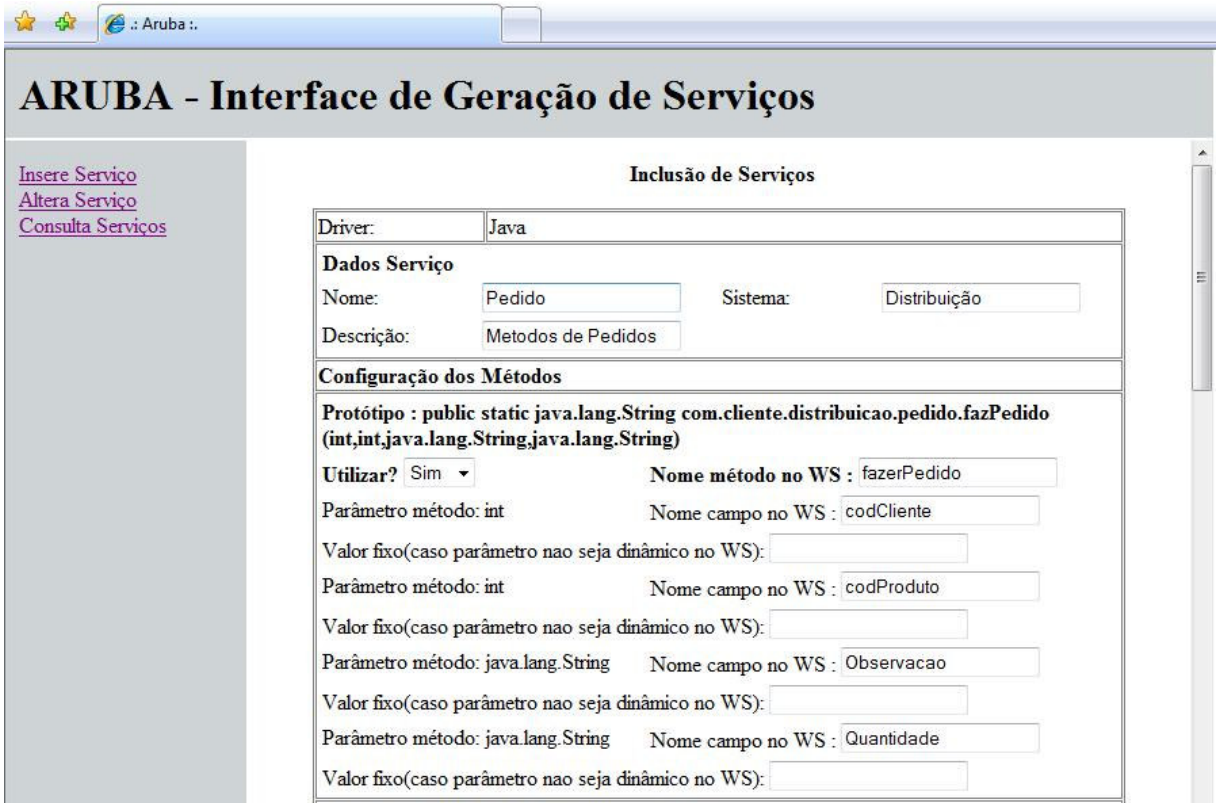

Figura 6.4 – Mapeamento dos métodos e parâmetros para geração do serviço.

Após configurados todos os métodos e pressionar o botão “Gerar Serviço”, os dados referentes ao serviço mapeado são gravados na base de dados e o arquivo XML de configuração e o *Web Services* são criados. A figura 6.5 apresenta a confirmação de geração do serviço.

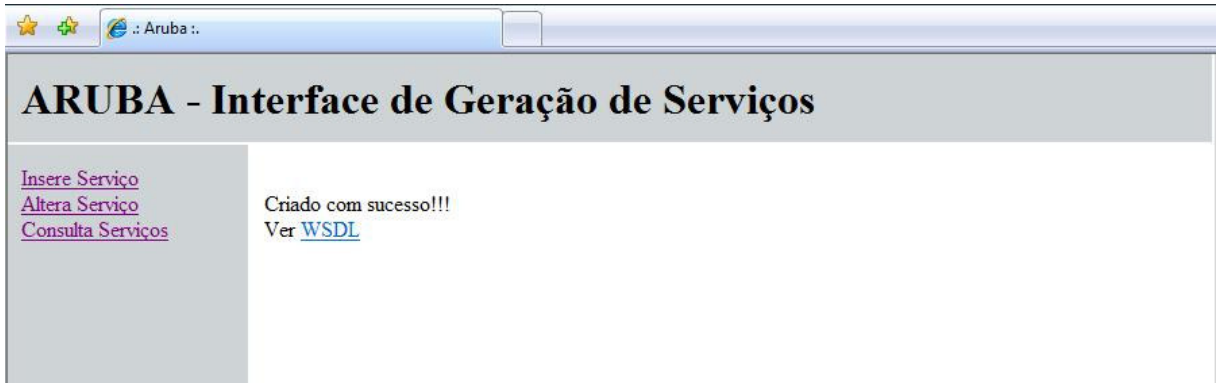

Figura 6.5 – Confirmação da geração do serviço.

O arquivo WSDL do *Web Services* pode ser visualizado pressionando no *link* ver “WSDL”. A figura 6.6 apresenta o arquivo gerado. O elemento *wsdl:definitions* traz, no

atributo *targetNamespace* (segunda linha da estrutura WSDL apresentada na figura 6.6), o endereço a partir do qual o *Web Services* poderá ser invocado. Neste caso, o endereço é *http://192.168.1.102/ws/Pedido.jws*.

Figura 6.6 – Wsdl do *Web Services* gerado.

O *Web Services* gerado corresponde às configurações efetuadas para disponibilização dos métodos da classe como serviço. A figura 6.7 apresenta o arquivo do *Web Services* gerado. Esse descreve os métodos mapeados, por exemplo, o método *fazerPedido*, possui os parâmetros *codCliente*, *codProduto*, *Observacao* e *Quantidade*, que correspondem aos quatro parâmetros originalmente mapeados (figura 6.7, linha 7 e 8).

```
1# import org.apache.axis.client.Call;
2# import org.apache.axis.client.Service;
3# import javax.xml.namespace.QName;
4# import org.apache.axis.encoding.XMLType;
5# import org.apache.axis.utils.Options;
6# public class Pedido {
7# public String fazerPedido(int codCliente,int codProduto,
8#   java.lang.String   Observacao,java.lang.String   Quantidade)   throws
Exception {
9# String retorno ="";
10# String endpoint = "http://localhost/ws/driverJava.jws";
11# Service  service = new Service();
12# Call     call    = (Call) service.createCall();
13# call.setTargetEndpointAddress( new java.net.URL(endpoint) );
14#   call.setOperationName(new   QName("http://localhost/ws/driverJava.jws?wsdl",
15# "callLegacyFunction"));
16# retorno = (String) call.invoke( new Object[] {
17# "/Program Files (x86)/Apache Software Foundation/Tomcat
18# 6.0/webapps/ws/xml/Pedido.xml",
19# "fazPedido",codCliente+","+codProduto+","+Observacao+","+Quantidade});
20# return (retorno);
21# }
22# public String cancelar(int codPedido) throws Exception {
23# String retorno ="";
24# String endpoint = "http://localhost/ws/driverJava.jws";
25# Service  service = new Service();
26# Call     call    = (Call) service.createCall();
27# call.setTargetEndpointAddress( new java.net.URL(endpoint) );
28#   call.setOperationName(new   QName("http://localhost/ws/driverJava.jws?wsdl",
29# "callLegacyFunction"));
30# retorno = (String) call.invoke( new Object[] {
31# "/Program Files (x86)/Apache Software Foundation/Tomcat
32# 6.0/webapps/ws/xml/Pedido.xml","cancelaPedido",codPedido});
33# return (retorno);
34# }
…
```

Figura 6.7 – Arquivo Pedido.jws gerado.

Observando o método *fazerPedido*, a variável *endpoint* (figura 6.7, linha 10) especifica o *driver* que será utilizado pelo *Web Service*, neste caso, a url do d*riverJava*. Os parâmetros para invocação do *Web Service driverJava.jws* são efetuadas através da criação de um objeto *call* (figura 6.7, linha 12), identificado por *Call call = (Call) service.createCall()*. Na chamada, através do método *invoke* (figura 6.7, linha 16), é enviado o caminho da arquivo de configuração do serviço gerado, o método e os parâmetros concatenados e separados por vírgula. Assim, o *driverJava* efetuará a comunicação com o sistema legado e o retorno é enviado para o método através da variável retorno.

A figura 6.8 apresenta o arquivo de configuração que é interpretado pelo *driverJava*. Neste arquivo, estão contidas as informações referentes a aplicação (*driver*, nomeServico, classe) e os métodos mapeados. Cada método é identificado com o nome e a máscara dos parâmetros.

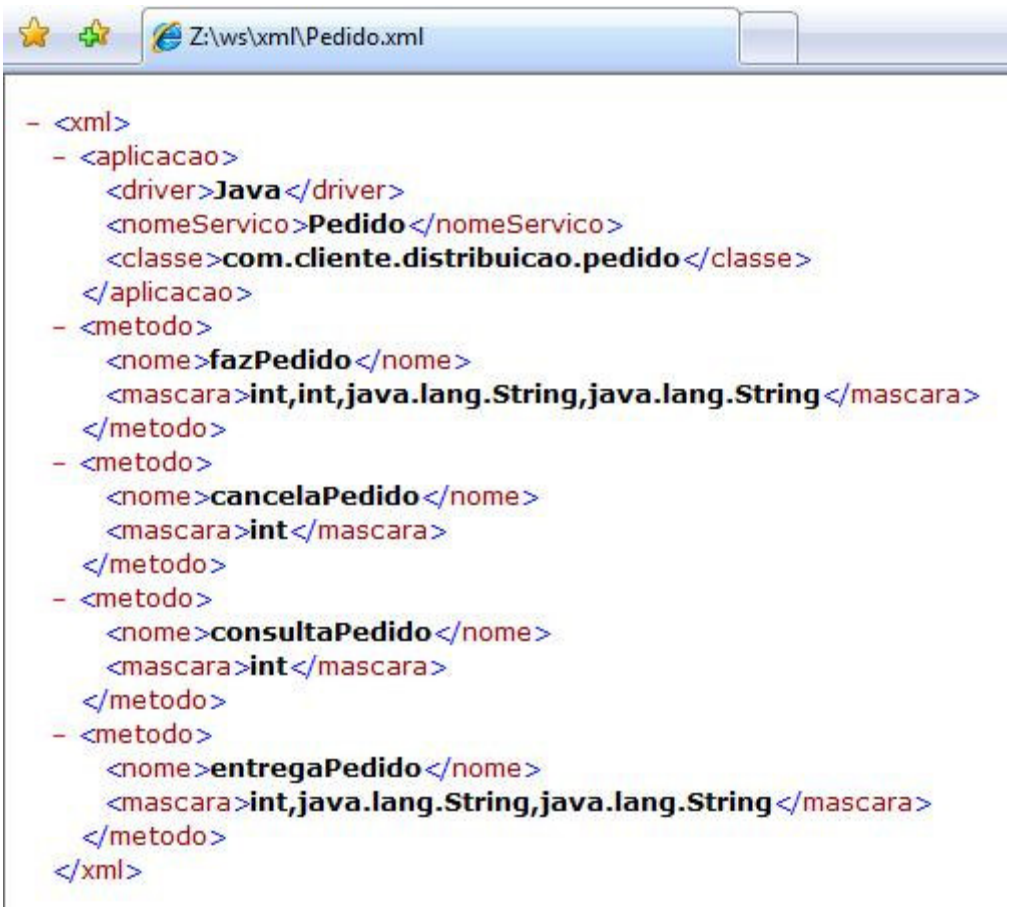

Figura 6.8 – Arquivo de configuração Pedido.xml gerado.

6.1.4 Comparativo: Implementação SOA x Aruba

Para visualização das diferenças entre a implementação efetuada em SOA e a implementação utilizando a arquitetura Aruba, as WBSs referentes aos dois projetos são apresentadas nas figuras 6.9 e 6.10.

Nome da tarefa	Duration	Resource Names	Cost	Work
Projeto	**129,91 days**		**R$ 98.505,57**	**2.377,22 hrs**
Pré-Planejamento	**0,5 days**		**R$ 710,90**	**12 hrs**
Planejamento	**6,88 days**		**R$ 4.387,77**	**95 hrs**
Especificação	**55,57 days**	**Analista de Sistemas**	**R$ 35.368,17**	**1.059,57 hrs**
Desenvolvimento	**30 days**	**Gerente de Projeto -**	**R$ 16.404,00**	**312 hrs**
Desenvolvimento conversor de unidades	14 days	Programador[75%]	R$ 4.200,00	84 hrs
Desenvolvimento dos Web Services	**14 days**		**R$ 4.200,00**	**84 hrs**
Sistema de Pedidos	6 days	Programador[75%]	R$ 1.800,00	36 hrs
Emissão de faturas	3 days	Programador[75%]	R$ 900,00	18 hrs
Conversor de unidades	2 days	Programador[75%]	R$ 600,00	12 hrs
Consulta Produto	3 days	Programador[75%]	R$ 900,00	18 hrs
Desenvolvimento das interfaces	**16 days**		**R$ 4.800,00**	**96 hrs**
Sistema de Pedidos	8 days	Programador[75%]	R$ 2.400,00	48 hrs
Emissão de faturas	3 days	Programador[75%]	R$ 900,00	18 hrs
Conversor de unidades	2 days	Programador[75%]	R$ 600,00	12 hrs
Consulta Produto	3 days	Programador[75%]	R$ 900,00	18 hrs
Testes	**36,72 days**	**Analista de Testes[7**	**R$ 25.323,46**	**562,22 hrs**
Entrega	**12,32 days**	**Gerente de Projeto -**	**R$ 8.459,17**	**206,92 hrs**
Encerramento	**5,5 days**	**Gerente de Projeto -**	**R$ 7.852,11**	**129,5 hrs**

Figura 6.9 – Project contendo a WBS do projeto original.

Nome da tarefa	Duration	Resource Names	Cost	Work
Projeto	**117,58 days**		**R$ 95.894,13**	**1.870,9 hrs**
Pré-Planejamento	**0,5 days**		**R$ 710,90**	**12 hrs**
Planejamento	**6,88 days**		**R$ 4.387,77**	**95 hrs**
Especificação	**55,57 days**		**R$ 35.368,17**	**615 hrs**
Desenvolvimento	**17,67 days**	**Gerente de Projeto -**	**R$ 13.792,56**	**250,27 hrs**
Mapeamento e configuracao dos legados para uso com Aruba	5,33 days	Arquiteto de Software	R$ 2.405,76	32 hrs
Desenvolvimento conversor de unidades	14 days	Programador[75%]	R$ 4.200,00	84 hrs
Desenvolvimento dos Web Services	**1,67 days**		**R$ 500,00**	**10 hrs**
Sistema de Pedidos	0,83 days	Programador[75%]	R$ 250,00	5 hrs
Emissão de faturas	0,33 days	Programador[75%]	R$ 100,00	2 hrs
Conversor de unidades	0,17 days	Programador[75%]	R$ 50,00	1 hr
Consulta Produto	0,33 days	Programador[75%]	R$ 100,00	2 hrs
Desenvolvimento das interfaces	**16 days**		**R$ 4.800,00**	**96 hrs**
Sistema de Pedidos	8 days	Programador[75%]	R$ 2.400,00	48 hrs
Emissão de faturas	3 days	Programador[75%]	R$ 900,00	18 hrs
Conversor de unidades	2 days	Programador[75%]	R$ 600,00	12 hrs
Consulta Produto	3 days	Programador[75%]	R$ 900,00	18 hrs
Testes	**36,72 days**	**Analista de Testes[7**	**R$ 25.323,46**	**562,22 hrs**
Entrega	**12,32 days**	**Gerente de Projeto -**	**R$ 8.459,17**	**206,92 hrs**
Encerramento	**5,5 days**	**Gerente de Projeto -**	**R$ 7.852,11**	**129,5 hrs**

Figura 6.10 – Project contendo a WBS do projeto com Aruba.

Em ambas as WBSs apresentadas, a coluna *Duration* representa o número em dias (úteis) de duração da tarefa. A coluna *Work* representa o tempo total, em horas, levado para se executar aquela tarefa. A coluna *cost* representa o custo para execução daquela tarefa, sendo obtido pelo resultado da multiplicação do número de horas com o custo do profissional envolvido (vide tabela 6.1).

No grupo de tarefas Desenvolvimento da figura 6.10, foi incluído uma nova tarefa denominada “Mapeamento e configuração dos legados para uso com Aruba”, essa tarefa contém o número de horas que foram necessários para mapear e configurar os métodos das classes legadas para geração dos serviços como *Web Services*.

A tarefa “Desenvolvimento conversor de unidades” consistiu na programação de um método que fizesse a conversão de unidades e medidas entre países e produtos. O método em questão foi utilizado neste estudo de caso sem nenhuma alteração e tanto o mapeamento quanto a geração do serviço foram efetuados utilizando a Aruba.

As tarefas pertencentes ao grupo de tarefas chamado “Desenvolvimento dos *Web Services*” referem-se aos tempos de desenvolvimento para os serviços em questão. Em todas estas tarefas foi utilizada a Aruba, não se utilizando os *Web Services* desenvolvidos no projeto original, sendo estes itens o principal objeto de comparação deste estudo de caso.

As tarefas pertencentes ao grupo de tarefas chamado “Desenvolvimento das Interfaces” referem-se aos tempos de desenvolvimento das *composite applications* que consomem os *Web Services* desenvolvidos. Para o estudo de caso utilizou-se as interfaces desenvolvidas no projeto original.

Assim como as WBSs, a tabela 6.6 apresenta as medições de custo e tempo efetuadas em tempo de desenvolvimento dos projetos original e com uso da Aruba, porém, com as informações consolidadas por tipo de tarefa.

Tabela 6.6 – Informações do projeto original x projeto com Aruba.

	Tempo total (horas)	**Tempo *Web Services* (horas)**	**Custo Total**	**Custo *Web Services***
Original	312	84	R$ 16.404,00	R$ 4.800,00
Com Aruba	250	10	R$ 13.792,00	R$ 500,00

Para salientar a diferença na comparação do tempo total do desenvolvimento, a figura 6.11 apresenta, em formato de gráficos, essa comparação.

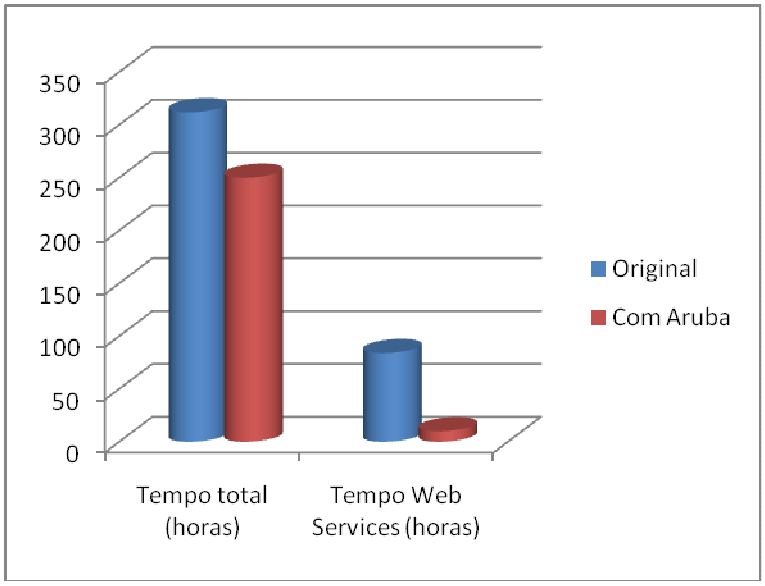

Figura 6.11 – Comparativo de tempo.

Para salientar a diferença na comparação do custo total do desenvolvimento, a figura 6.12 apresenta, em formato de gráficos, essa comparação.

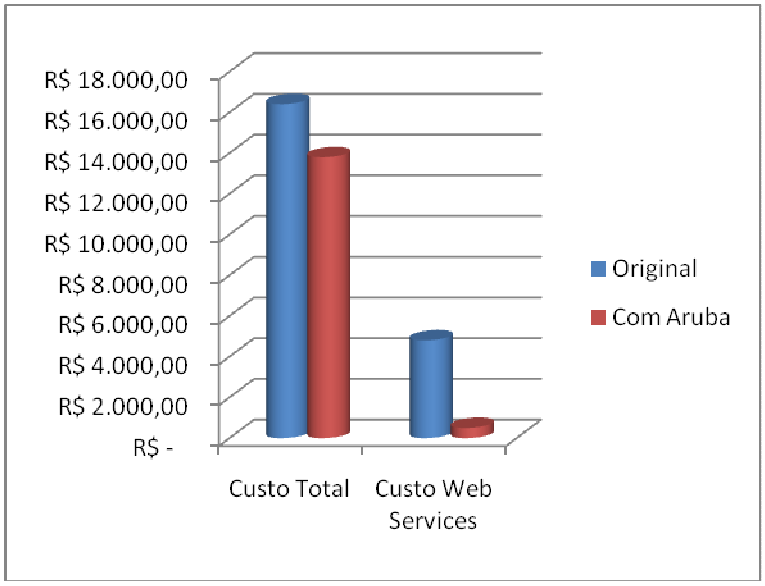

Figura 6.12 – Comparativo de custo.

6.2 Conclusões e considerações

Comparando a implementação original do projeto com o uso da Aruba, pode-se constatar que:

- A quantidade de horas necessárias para desenvolvimento dos *Web Services* cai na proporção de 8,4 para 1 (horas), valor esse obtido pela divisão das 84 horas necessárias no projeto original pelas 10 horas necessárias com o uso da Aruba;
- Em relação à camada de regras de negócios não houve alterações, visto que na geração dos *Web Services* com Aruba, utilizou-se as classes desenvolvidas no projeto original;
- Alterações nas classes do projeto original refletem diretamente na geração dos serviços, sem necessidade de recodificação, já que a interface de configuração da Aruba possibilita a alteração dos serviços criados;
- O número consolidado de horas de desenvolvimento foi reduzido em aproximadamente 20%, valor esse obtido comparando-se as 312 horas necessárias no projeto original com as 250 horas necessárias com o uso da Aruba;
- O custo financeiro consolidado de desenvolvimento foi reduzido em aproximadamente 16%, custo obtido comparando-se os R$ 16.404,00 necessários para o desenvolvimento no projeto original contra os R$ 13.792,00 necessários com o uso da Aruba (considerando os custos apresentados na tabela 6.1);
- A integração com o Oracle *Forms* e o ERP foram efetuadas com APIs específicas (não compatíveis com a Aruba) no projeto original, não fosse isso, o potencial ganho seria maior tendo em vista que um sistema, por exemplo, em Java, permitiria o uso da Aruba na reutilização das classes do sistema legado;
- Optou-se por fazer uso de um sistema corporativo real e não um sistema fictício para geração do estudo de caso, assim, algumas limitações deste sistema foram respeitadas, como, por exemplo, a observação no item anteriormente citado.

7 CONCLUSÃO

Este capítulo apresenta uma comparação entre os trabalhos relacionados versus a arquitetura Aruba, assim como, as limitações deste trabalho e os trabalhos futuros.

7.1 Comparativo dos trabalhos relacionados versus Aruba

Para elucidar qual a contribuição deste trabalho sob o contexto de SOA, principalmente sob o viés das ferramentas já existentes, a tabela 3.1 será repetida abaixo, agora incluindo a arquitetura Aruba no comparativo.

Tabela 7.1 – Tabela comparativa.

	Smart	*Encapsulamento de código legado*	*IBM WebSphere*	*SAP NetWeaver*	*Microsoft Biztalk Server*	*Oracle SOA Suite*	Aruba
Acesso não intrusivo	Não	Não	Não	Não	Não	Não	Sim
Geração de *Web Services*	Não	Sim	Sim	Sim	Sim	Sim	Sim
Catálogo de serviços	Não	UDDI	UDDI	UDDI ou diretório próprio	UDDI ou diretório próprio	UDDI	UDDI
Uso de adaptadores	Não	Não	Sim	Sim	Sim	Sim	Sim

Sob as várias aplicações e funcionalidades possíveis para uma ferramenta, IDE, ambiente ou arquitetura voltada a SOA, o foco da arquitetura Aruba está nos quatros critérios da tabela 7.1, “Acesso não intrusivo”, “Geração de *Web Services*”, “Catálogo de Serviços” e “Uso de adaptadores”.

Por motivos que variam desde desempenho até estratégia comercial, as ferramentas IBM *WebSphere*, SAP *NetWeaver*, Microsoft *BizTalk Server* e Oracle *SOA Suite,* assim como

as duas técnicas apresentadas, baseiam-se em alterações nos sistemas legados. Por outro lado a arquitetura Aruba fornece "Acesso não intrusivo" para um sistema legado, ou seja, não altera o código existente para disponibilizar uma regra de negócio como serviço. A arquitetura Aruba faz uso de adaptadores nomeados de "*drivers*", cuja configuração é feita em uma interface sem necessidade de programação, que são responsáveis por processar uma requisição, acessar um sistema legado e retornar uma resposta a essa requisição.

Quanto à "Geração de *Web Services*", a arquitetura Aruba os gera para servir como acesso aos serviços criados a partir dos sistemas legados, o que é feito através de arquivos de configuração baseados no formato XML. Isso muito se assemelha as ferramentas citadas que também geram *Web Services,* porém baseados em configurações específicas das IDEs de cada ferramenta, não ficando abertas para alterações em um formato de padrão aberto como o XML.

Os *Web Services* (que na verdade são a interface para os serviços da arquitetura Aruba) são armazenados num repositório nomeado de "Catálogo de Serviços". Analisando este critério, verifica-se que a Aruba segue o formato padrão para publicação e descoberta de *Web Services*, o UDDI. Porém, Aruba também disponibiliza uma funcionalidade que permite que os serviços registrados no repositório de dados sejam visualizados.

Além dos critérios citados na comparação, a utilização efetiva dessa arquitetura pode trazer outras vantagens:

1. Geração de *Web Services* a partir de classes Java e .Net já existentes, sem necessidade de codificação;
2. Possibilidade de mapeamento de mais de uma classe como serviço (*Web Services*), sem a necessidade de alterar o código já existente;
3. Interface Web para configuração e geração de serviços, não sendo necessária a instalação de nenhuma ferramenta na máquina do usuário;
4. Geração de *Web Services* feita por uma interface comum, independente do *driver* utilizado;
5. No caso do uso do *driver* DDE, permite que telas de sistemas mais antigos (ex. Cobol) sejam mapeadas na tentativa de permitir seu uso com *Web Services*;

6. Além do UDDI, permite o uso de um repositório centralizado (banco de dados) para cadastro e gerência dos serviços. A alteração desse cadastro automaticamente altera seu respectivo *Web Services* de forma transparente para o usuário;
7. Tempo de desenvolvimento de *Web Services* a partir de regras de negócios implementadas em classes nas linguagens Java e .Net reduzido se comparado ao tempo de desenvolvimento necessário a um programador.

7.2 Limitações

A Aruba é uma arquitetura focada no mapeamento de métodos legados e na geração destes como *Web Services*. Devido a arquitetura e o objetivo proposto, algumas limitações são identificadas:

1. Acesso ao sistema legado: o acesso só é garantido quando mapeado corretamente na criação do serviço;
2. *Drivers*: a arquitetura disponibiliza três tipos de *drivers.* Portanto, para aplicações desenvolvidas em linguagens que não sejam Java, .Net ou que não tenham interface com o usuário, uma customização deverá ser efetuada;
3. *Driver* Java ou .Net: não é possível combinar mais de um método das classes em um método do *Web Service*;
4. *Driver* Java ou .Net: a arquitetura não é compatível com dependências ou sessões (autenticação, variáveis de ambiente, etc);
5. *Driver* Java ou .Net: métodos que usem parâmetros ou retornos customizados (não mapeáveis a um tipo XSD) não funcionarão corretamente;
6. *Driver* DDE: é necessária uma máquina dedicada para este, de forma que uma sessão de usuário possa ser simulada com o sistema;
7. *Driver* DDE: funções que requererem mais de uma tela para serem executadas (ex. um cadastro longo) não poderão ser utilizadas pela arquitetura;
8. Os sistemas legados a serem mapeados na arquitetura devem estar bem documentados, ou um esforço de documentação deve ser feito a priori do uso da arquitetura, de forma que, as classes, métodos e parâmetros sejam conhecidos para que a geração de serviços seja possível.

7.3 Trabalhos Futuros

Para complementação do trabalho, podem ser considerados como possíveis trabalhos futuros:

1. Criação de *drivers* que atendam outras linguagens de programação;
2. Mecanismos de suporte à segurança e autenticação, por exemplo, *WS-Security*;
3. Composição de métodos na geração do serviço (*Web Services*) a partir de vários métodos dos sistemas legados;
4. Implementar mecanismos para suporte a sessão ou dependência entre as classes (Java e .Net), estendendo o uso da Aruba além da SOA;
5. Uso da arquitetura em outros cenários que exijam o uso do *driver* DDE para avaliação de seus benefícios além do estudo de caso apresentado.

8 REFERÊNCIAS

[Bastos, 2007] Bastos, J. S. Y. Utilização de fontes de informação por executivos de setor de tecnologia da informação no Brasil. Revista Fonte. ISSN 1808-0715. N.5, January, 2007.

[Banerjee, 2007] Banerjee, A. Building Office Business Applications. The architecture journal. Input for Better Outcomes. Journal 10. Microsoft. ARC 098-107185.

[Bergey, 2002] Bergey, J.; O'Brien, L.; Smith, D. Using the Options Analysis for Reengineering (OAR) Method for Mining Components for a Product Line, 316-327. Software Product Lines: Proceedings of the Second Software Product Line Conference (SPLC2). San Diego, CA, August 19-22, 2002. Berlin, Germany: Springer, 2002.

[Brown et al., 2002] Brown, A.; Johnston, S.; Kelly, K.. Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications. A Rational Software White Paper. 2002.

[CIO, 2006] Chief Information Office. Inovar é preciso. September, 2006. Disponível em: <http://cio.uol.com.br/gestao/2006/09/19/idgnoticia.2006-09-19.6666023153/IDGNoticia_view>. Acesso em: 12 de dez. 2007.

[Chappell, 2004] Chappell, D.. Understanding BPM Servers. David Chappell & Associates. Microsoft Corporation. October, 2004.

[Chen et al., 2006] Chen, Y., Fan, C., Tsai, W.T., Paul, R., Chung, J..Architecture Classification for SOA-Based Applications. Proceedings of the Nineth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, 2006.

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

[Erl, 2006] Erl, T. Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall, 2006.

[Gartner, 2003] Gartner Research. Service-Oriented Architecture Scenario. April, 2003. Disponível em: <http://www.gartner.com/resources/114300/114358/114358.pdf>. Acesso em: 12 de dez. 2007.

[Graham et al., 2002] Graham, S.; Simeonov, S.; Boubez, T.; Davis, D.; Daniels, G.; Yuichi, N. and Neyama, R. Building Web Services with Java: making sense of XML, SOAP, WSDL, and UDDI. Indianápolis: Sams, 2002. 581p.

[Hansen, 2003] Hansen, R. P.. 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.

[Heuser, 2004] Heuser, L.. Enterprise Services Architecture & Semantic Web Services. SAP Research. SAP Group. 2004.

[IBM, 2005] IBM Corporation. IBM SOA Foundation: providing what you need to get started with SOA. Service oriented architecture solutions. White Paper, 2005.

[IBM, 2006] IBM Corporation. Increasing IT flexibility with IBM WebSphere ESB software.White Paper, 2006.

[IDC, 2007] International Data Corporation. Releases 2007, January 24th, 2007. Disponível em: <http://www.idclatin.com/news.asp?ctr=bra&year=2007&id_release=855>. Acesso em: 12 de dez. 2007.

[Keen et al., 2004] Keen, M.; Acharya, A.; Bishop, S.; Hopkins, A.; Milinski, S.; Nott, C.; Robinson, R.; Adams, J.; Verschueren, P.. Patterns: Implementing a SOA Using an Enterprise Service Bus. IBM Redbooks, First Edition, July, 2004.

[Kevin, 2006] Kevin, F.. Workflow in Application Integration. The architecture journal. Input for Better Outcomes. Journal 7. Microsoft. ARC 098-105109.

[Kreger, 2001] Kreger, H. Web Services Conceptual Architecture. IBM Software Group, May 2001.

[Lewis et al., 2005a] Lewis, G.; Morris, E.; O'Brien, L.; Smith, D.; Wrage, L.. SMART: The Service-Oriented Migration and Reuse Technique. Technical Note CMU/SEI-2005-TN-029, September, 2005.

[Lewis et al., 2005b] Lewis, G.; Morris, E.; Smith, D.. Migration of Legacy Components to Service-Oriented Architectures. The DOD Software Tech, Vol.8, n.3, October, 2005.

[Manolescu and Lublinsky, 2007a] Manolescu, D.; Lublinsky B.. Draft Service-Oriented Architecture Defined, do livro Enterprise Patterns - Services, Orchestration and Beyond, para ser publicado por Morgan-Kaufmann. Disponível em <http://orchestrationpatterns.com>. Acesso em: 22 de mar. 2007.

[Manolescu and Lublinsky, 2007b] Manolescu, D.; Lublinsky, B.. Draft Pattern: Movement: Service-Oriented Decomposition, do livro Enterprise Patterns - Services, Orchestration and Beyond, para ser publicado por Morgan-Kaufmann. Disponível em <http://orchestrationpatterns.com>. Acesso em: 12 de dez. 2007.

[Microsoft, 2006a] Microsoft Corporation. Learn About Service Oriented Architecture (SOA). December, 2006.

[Microsoft, 2006b] Microsoft Corporation. Serviços da Web e a plataforma Microsoft. Microsoft White Paper. August, 2006.

[Newcomer, 2002] Newcomer, E. Understanding Web Services. Independent Technology Guides. Dadid Chappell, Series Editor. 2002.

[OASIS, 2006] OASIS Reference Model for Service Oriented Architecture V 1.0. August 2, 2006. Disponível em <http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm>. Acesso em: 12 de dez. 2007.

[Oellermann, 2001] Oellermann Junior, W.L. Architecting Web Services. New York: Apress, 2001. 654p.

[Oracle, 2005] Strategies for SOA Success. Ziff Davis Media Custom Publishing. December, 2005.

[Oracle, 2006] Oracle SOA Suite. Quick Start Guide 10g (10.1.3.1.0). September, 2006.

[Parnas, 1972] Parnas, D. L. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12). December, 1972.

[Rheinheimer, 2004] Rheinheimer, L. R. WSAgent: Um Agente Baseado em Web Services para promover a Interoperabilidade entre Sistemas Heterogêneos no Domínio da Saúde. Tese de Mestrado pela Unisinos, 2004.

[Rogers and Hendrick, 2005] Rogers, S.; Hendrick, S. D.. Oracle Builds Comprehensive SOA Platform. IDC White Paper. January, 2005.

[Roy and Ramanujan, 2001] Roy, J.; Ramanujan, A. Xml Shema Language: Taking XML to The Next Level, IT Pro, vol. 3, No. 2, March/April 2001, pp. 37-40.

[Sampaio, 2006] Sampaio, C. SOA e Web Services em Java. Rio de Janeiro: Editora Brasport 2006.

[SAP, 2006] SAP Group. SAP NetWeaver: Using IT practices to bridge the worlds of business and IT. November, 2006.

[SAP, 2007] SAP Group. Transform your organization with enterprise service-oriented architecture. January, 2007.

[Schimidt, 2003] Schimidt, R. Composite Applications for the Enactment of Dynamic Inter-Organizational Business Processes, in Proceedings of the 1st Int. Workshop "Component Based Business Information Systems Engineering" (Internal Conference on Object Oriented Information Systems). Genova, September 2nd, 2003.

[Sehmi, 2006] Sehmi, A. Schwegler, B.. Modeling and Messaging for Connected Systems. The architecture journal. Input for Better Outcomes. Journal 7. Microsoft. ARC 098-105109.
[W3C, 2002] Web Services architecture requirements: W3C working group note. February. 2004. Disponível em: <http://www.w3.org/TR/wsa-reqs>. Acesso em: 12 de dez. 2007.

[Sneed, 1996] Sneed, H.; Erdoes, K. Extracting Business Rules from Source Code, Proc. of 4th IWPC-1996, IEEE Computer Society, Berlin, March 1996, p. 240.

[Sneed, 2006] Sneed, H. M. Wrapping Legacy Software for Reuse in a SOA. Third GI-Workshop XML4BPM XML Integration and Transformation for Business Process Management. Passau, Germany, February 22th, 2006.

[W3C, 2007] Web Services architecture: W3C working group note. February. 2004. Disponível em: <http://www.w3.org/TR/ws-arch>. Acesso em: 12 de dez. 2007.