UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
ESCOLA DE INFORMÁTICA APLICADA
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Uma Abordagem Prática para Desenvolvimento de Web Services com Contract-First

Nome do autor:
Sandro Pinheiro Lopes
Marcos de Barros Mele

Nome do Orientador:
Leonardo Guerreiro Azevedo

Julho/2012

Uma Abordagem Prática para Desenvolvimento de Web Services com Contract-First

Projeto de Graduação apresentado à Escola de Informática Aplicada da Universidade Federal do Estado do Rio de Janeiro (UNIRIO) para obtenção do título de Bacharel em Sistemas de Informação

Nome dos autores:

Sandro Pinheiro Lopes

Marcos de Barros Mele

Nome do Orientador:

Leonardo Guerreiro Azevedo

Uma Abordagem Prática para Desenvolvimento de Web Services com Contract-First

Aprovado em ____/_____/_____

BANCA EXAMINADORA

_____

Prof. Leonardo Guerreiro Azevedo, D.Sc. (UNIRIO)

_____

Prof. Mácio de Oliveira Barros, D.Sc. (UNIRIO)

_____

Prof. Alexandre Luis Correa, D.Sc. (UNIRIO)

Rio de Janeiro, ____ de _____ de _____

_____

Sandro Pinheiro Lopes

_____

Marcos de Barros Mele

Agradecimentos

Agradecemos primeiramente a Deus pelo desenvolvimento e orquestração de tudo.

Às nossas famílias não somente pelo incentivo sempre bem-vindo e necessário, mas pela paciência pelas muitas ausências e entendimento da importância desse estágio de nossas vidas.

Ao nosso orientador Leonardo Guerreiro Azevedo por ter nos apresentado com tanta paixão esse universo que é SOA. Sua atenção, dedicação e competência foram cruciais para elaboração deste trabalho.

Ao Luiz Felipe Velloso que nos concedeu seu servidor para que nós pudéssemos desenvolver em um ambiente controlado e centralizado. Ao Michael Ramos pela ajuda competente que nos deu em diversas tecnologias que utilizamos em nosso projeto.

Aos nossos amigos que, como nossas famílias, sentiram nosso afastamento, mas mesmo sem o laço sanguíneo entenderam a necessidade e torceram por nós.

Ao corpo docente da Uniriotec pela competência em nos proporcionar uma formação de qualidade e diferenciada.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

SOA – Service-Oriented Architecture

BPM – Business Process Management

BPEL – Business Process Execution Language

ESB – Enterprise Service Bus

SOAP – Simple Object Access Protocol

WSDL – Web Services Description Language

XSD – XML Schema Definitions

XML – eXtensible Markup Language

RPC – Remote Procedure Call

JPA – Java Persistence API

EJB – Enterprise Java Beans

JEE – Java Enterprise Edition

JAX-WS – Java Architecture for XML Web Services

JAXB – Java Architecture for XML Binding

RESUMO

A arquitetura orientada a serviços (SOA) propõe comunicação e interoperabilidade entre sistemas heterogêneos, sejam eles legados ou novos. Entretanto, mesmo usando uma abordagem SOA, a integração entre sistemas é um desafio. Existem propostas de padrões de desenvolvimento de serviços para que os mesmos se tornem interoperáveis. Na prática, a abordagem de desenvolvimento mais utilizada no mercado é a geração automática do contrato após o desenvolvimento (code-first). Contudo, essa abordagem gera dificuldades de manutenção e consumo do serviço em longo prazo, tais como: impossibilidade de utilização de tipos de dados centralizados, alta probabilidade de mudança no contrato no caso de manutenção na implementação do serviço e impossibilidade de versionamento. Uma melhor abordagem é construir o contrato do serviço primeiro e a partir do contrato implementar o código aderente a este contrato. Este trabalho apresenta um estudo de caso com aplicação de abordagem de desenvolvimento contract-first, a fim de garantir interoperabilidade, aumento do grau de reuso e durabilidade, facilidade de governança e redução do acoplamento com uma determinada tecnologia.

Palavras-chave: SOA, Contract-First, Interoperabilidade, Basic Profile (BP), WSDL, Web Services e Centralização de Esquema.

Capítulo 1: Introdução

O objetivo deste capítulo é contextualizar os conceitos e fundamentações teóricas utilizadas neste trabalho. Além disto, serão apresentados a motivação, o objetivo do trabalho e a estrutura dos capítulos.

1.1 Contextualização

Segundo Josuttis [2007], SOA (Service-Oriented Architecture) possibilita a comunicação entre sistemas desenvolvidos em linguagens de programação distintas e sendo executados em sistemas operacionais diferentes. Esse cenário é muito comum em empresas de médio e grande porte que possuem uma infra-estrutura de TI complexa.

A definição de Erl [2005] identifica SOA como um paradigma arquitetural que disponibiliza funcionalidades de aplicações de maneira autocontida como serviços. Serviços são componentes de software que representam uma atividade do negócio [Hewitt 2009]. Em geral, serviços são desenvolvidos utilizando a tecnologia de *web services* e são responsáveis pela comunicação e interoperabilidade entre os sistemas. No mundo de *web services*, serviços são descritos utilizando a linguagem WSDL (*Web Services Description Language*) e a comunicação é feita de forma padronizada utilizando troca de mensagens escritas segundo o protocolo SOAP (Simple Object Access Protocol).

Papazoglou[2007] apresenta os dois principais papéis em uma arquitetura SOA: provedor e consumidor. Provedor realiza todos os passos do desenvolvimento do serviço, desde a identificação da necessidade ao provimento do mesmo. Após a publicação do serviço, os consumidores estão aptos a consumi-los através da interface disponibilizada. Existe ainda um terceiro papel, do broker, responsável por realizar a manutenção dos registros de serviços publicados.

Em um complexo sistema corporativo, é necessária uma infraestrutura em que consumidores e provedores de serviços se comuniquem de forma eficiente e segura e que não seja ponto-a-ponto, reduzindo acoplamento entre consumidores e provedores, i.e., é necessário haver mediação de comunicação consumidor-provedor.

Conforme descrito por Josuttis [2007], a principal infra-estrutura para mediação da comunicação entre consumidores e provedores é chamada de *Enterprise Service Bus* (ESB).

Josuttis [2007] apresenta a ligação entre SOA e a utilização de uma gestão de processos de negócio, e como a utilização de BPM (Business Process Management) representa um papel importante em SOA, uma vez que serviços são partes de processos de negócio. O modelo de processos estabelece a seqüência de atividades que compõem o processo de negócio. O desenho de atividades e fluxo das mesmas apresentam informações importantes sobre o negócio tais como produtos e insumos, se são executadas por sistemas ou executadas manualmente ou apoiadas por sistemas, papéis que executam a atividade etc. Enfim, o modelo de negócio apresenta um conjunto de informações sobre as quais é possível realizar uma análise sistemática para identificação de serviços e, posteriormente, a análise desses serviços. Essa sistematização na identificação e análise foram propostas respectivamente por Azevedo *et al*. [2009a, 2009b, 2009c, 2009d, 2009e, 2011].

1.2 Motivação

Souza *et al*. [2010] realizaram a identificação e análise de serviços a partir de um modelo do processo de negócio para tratar análise de crédito, seguindo as heurísticas propostas por Azevedo *et al*.[2009a, 2009b, 2009c, 2009e, 2011].

Em Souza *et al.*, [2011] foram definidos, dentre a listagem obtida após análise dos serviços candidatos, quais serviços seriam implementados. Este trabalho dará continuidade ao estudo de caso relativo ao processo de análise de crédito, seguindo o ciclo de vida apresentado por Gu e Lago [2007] nas fases projeto e implementação dos serviços obtidos.

Como em SOA a principal comunicação entre provedor e consumidor é estabelecida através do contrato do serviço, cada serviço desenvolvido terá primeiro seu contrato estabelecido, evitando que mudanças e tendências durante o desenvolvimento afetem os consumidores do serviço. Alguns padrões e convenções serão abordados, a fim de garantir interoperabilidade, redução de acoplamento, aumento de reusabilidade e melhoria em governança dos serviços.

1.3 Objetivo do trabalho

O objetivo deste trabalho é se aprofundar na fase de desenvolvimento do ciclo de vida de serviços presente no modelo de ciclo de vida de serviços orientado a *stakeholder* de Gu e Lago [2007] utilizando contract-first, que desenvolve primeiro os artefatos relativos ao contrato de serviço antes de sua codificação. A utilização desse padrão de desenvolvimento auxilia a elaboração de serviços mais independentes, coesos, eficientes e aderentes ao negócio. Concomitantemente à utilização de contract-first, serão utilizados padrões e boas práticas de desenvolvimento de serviços presentes na especificação do Basic Profile (BP) que visam garantir interoperabilidade, redução de impacto de mudanças e facilidade de manutenção de serviços.

Como o foco do trabalho está no desenvolvimento do serviço em si, os resultados apresentados por Souza *et al.* [2011] na identificação e análise de serviços serão empregados como ponto de partida para este trabalho. Isto permitirá uma análise mais detalhada de tendências entre a fase de análise e a de implementação e, caso necessário, realizar proposta de melhoria da fase de análise e projeto para se adequar melhor à realidade da implementação e composição de serviços.

A abordagem escolhida para implementação de serviços foi web services e a implementação da composição de serviços utilizando injeção direta no serviço composto. Os contratos dos serviços foram desenvolvidos utilizando WSDL, que é linguagem padrão de definição de web services; enquanto a API JAX-WS foi escolhida para desenvolvimento de web services devido a sua robustez e praticidade por permitir uso de *annotations* (padrão de mapeamento por anotação de classes Java).

1.4 Estrutura do trabalho

O restante deste trabalho está dividido da seguinte forma. O capítulo 2 apresenta os conceitos necessários ao entendimento do trabalho como: SOA; Web services; WSDL; XSD; ESB; BPM; ciclo de vida de serviços, métodos para identificação, análise e projeto de serviços; e orquestração e coreografia. O capítulo 3 discorre sobre o conceito de contrato de serviço, apresenta o Basic Profile (BP), que é o

principal produto do consórcio Web Service Interoperability (WS-I) e define padrões de desenvolvimento de serviços com maior interoperabilidade. Por fim, traz um comparativo das abordagens contract-first e code-first. O capítulo 4 trata do desenvolvimento dos serviços identificados utilizando o framework JAX-WS 2.2 e outras especificações J2EE. Por fim, o capítulo 5 apresenta a conclusão do trabalho.

Capítulo 2: Conceitos

2.1 SOA

SOA é o acrônimo para Service-Oriented Architecture (Arquitetura Orientada a Serviço). Entretanto, SOA não é uma arquitetura concreta, mas um sistema de valores que conduz a uma arquitetura concreta de software. Desta forma, não é possível comprar SOA, como se compra um *framework*, com a intenção de garantir precisão e qualidade no desenvolvimento de um projeto [Josuttis, 2007]. Erl [2005] define a Arquitetura Orientada a Serviços (SOA) como sendo uma arquitetura onde funcionalidades de software são disponibilizadas como serviços, facilitando assim a comunicação e interoperabilidade entre aplicações, além de criar uma infraestrutura de TI mais flexível e alinhada com o negócio.

Serviços são pedaços de funcionalidades auto-contidas, que possuem interfaces expostas e que são invocados via mensagens [Josuttis, 2007]. Estas funcionalidades podem fazer parte de um ou mais processos e serviços podem ser desenvolvidos em qualquer tecnologia e em qualquer plataforma que permita comunicação interoperável.

Serviços podem representar funcionalidade simples ou complexas. Um exemplo de serviço simples é aquele para armazenamento ou recuperação de dados. Por outro lado, um exemplo de serviço complexo seria aquele que exige uma solução lógica mais sofisticada como, por exemplo, execução de diversas trocas de mensagens, mecanismos de transação ou compensação e execuções paralelas. A qualidade de projetos baseados no padrão SOA está diretamente relacionada com o conhecimento dos projetistas sobre esse paradigma e, obviamente, sobre as especificações relacionadas ao projeto a ser desenvolvido.

"Os grandes sistemas utilizam plataformas diferentes, linguagens de programação diferentes (e paradigmas de programação) e até *middleware* diferente. Eles são um conjunto de mainframes, clientes SAP, bancos de dados, aplicações J2EE, pequenos motores de regras, e assim por diante. Em outras palavras, eles são heterogêneos" [Josuttis, 2007]. Um exemplo ilustrativo é apresentado na Figura 1,

onde há uma rede de comunicação entre diversos servidores, sejam eles sistemas em diferentes linguagens, bancos de dados distribuídos ou aplicações em nuvem, todos trocando informações entre si.

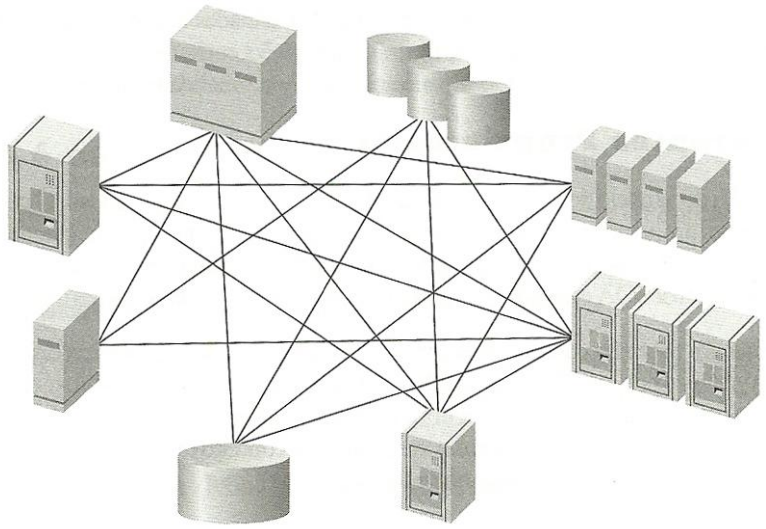

Figura 1 – Sistemas Heterogêneos [Josuttis, 2007]

Em um ambiente heterogêneo como o descrito por Josuttis, é possível e provável que um requisito de software possa estar sendo utilizado em uma solução computacional com proprietários e tecnologias diferentes. Em cenários como esses, a necessidade de integração de diferentes tipos de sistemas se torna vital para o sucesso da empresa.

Diferentemente de outras abordagens, SOA aceita essa heterogeneidade reconhecendo, organizando e utilizando essas capacidades distribuídas de diferentes proprietários [OASIS, 2006] através de uma arquitetura baseada em padrões. Essa arquitetura fornece uma poderosa estrutura para adequar e combinar essas capacidades distribuídas.

Ambientes heterogêneos são difíceis de gerenciar sem essa abordagem padronizada. Assim, SOA possibilita o desenvolvimento de soluções mais ágeis para atender às demandas do mercado. Afinal, tão importante quanto desenvolver um software de qualidade é desenvolvê-lo em um tempo que possa acompanhar o ritmo e necessidades do mercado e, assim, esteja disponível em tempo hábil para agregar valor ao negócio da empresa. Muito dessa agilidade e flexibilidade se deve à interoperabilidade, reutilização e orquestração de serviços.

2.2 Web services

Web service é um conjunto de padrões que possibilita a interoperabilidade entre sistemas heterogêneos. Esses padrões definem os protocolos utilizados para comunicação e formatos de interface para especificação e contratos de serviços [Josuttis, 2007].

Web services são auto-suficientes, auto-descritivos, independentes de linguagem e fracamente acoplados. Eles possuem natureza modular e podem ser publicados, localizados e invocados através da World Wide Web [Iyengar *et al.*, 2008] utilizando, principalmente, o protocolo HTTP.

"A tecnologia de Web Services fornece uma maneira eficiente para compartilhar lógica de aplicação em múltiplas máquinas que executam sistemas operacionais diferentes e com diferentes ambientes de desenvolvimento. Para isso, Web Services utilizam SOAP, WSDL, XML Schema e outras tecnologias baseadas em XML, fornecendo uma abordagem baseada em padrões para superar as diferenças de plataforma e linguagem." [Vasiliev, 2007]. Dessa forma, como descrito por Vasiliev, existem alguns padrões baseados em XML que possibilitam a interoperabilidade e demais características fundamentais de um web service e, dentre eles, o entendimento do padrão WSDL é imprescindível para este trabalho.

2.3 WSDL

Web Services Description Language (WSDL) é a linguagem que fornece um vocabulário XML, razoavelmente simples, para descrever o serviço, as operações disponíveis, os formatos de mensagens e como acessá-lo. Atualmente, existem duas versões de WSDL utilizadas para definição de serviços: WSDL 1.1 [Christensen *et al.*, 2001] e WSDL 2.0 [Chinnici *et al.*, 2007].

A versão 1.1, apesar de algumas carências iniciais em termos de documentação de recursos, de ser menos intuitiva e flexível é a mais difundida no mercado e, por isso, é a que contém maior número de ferramentas de apoio ao desenvolvimento de web services [Sosnoski, 2011].

A versão 2.0 do WSDL foi desenvolvida para ser mais simples, flexível e melhor documentada quanto a seu propósito e recursos. Em 2007, o W3C substituiu tecnicamente a versão 1.1 endossando a versão 2.0 do WSDL. Entretanto, esta versão ainda é pouco difundida [Sosnoski, 2011].

Na prática, isso significa que serviços desenvolvidos utilizando a versão 1.1 tem maior capacidade de interoperabilidade. Por isso, a Web Services Interoperability Organization (WS-I) utilizou o WSDL 1.1 para definir boas práticas para serviços web em seu Basic Profile (BP).

Um arquivo WSDL 1.1 pode ser classificado em três partes distintas que correspondem à definição abstrata, definição concreta e documentação do serviço. Cada elemento de um arquivo WSDL corresponde a uma dessas três classificações. Os elementos *types, message, portType e operations* são classificados como elementos de interface e representam a definição abstrata do serviço. Os elementos *binding* e *services* são classificados como elementos de programação e representam a definição concreta do serviço. O elemento *documentation* é classificado como elemento de documentação e representa a documentação do web service para torná-lo compreensível e acessível aos usuários [Erl, 2004]. A Tabela 1 apresenta os elementos existentes em um arquivo WSDL 1.1, versão utilizada neste trabalho, e seu objetivo. A Figura 2 mostra um exemplo de WSDL na versão 1.1.

Tabela 1 – Partes do WSDL

Elemento	Descrição
Definitions	Elemento raiz do arquivo WSDL
Types	Define os tipos de dados (XSD Language) utilizados pelo serviço.
portType	Representa a interface do serviço, podendo conter múltiplas operações.
Operation	Representa uma função do web service.
Message	Representa coleções de parâmetros de entrada e saída.
Part	Representa dados usados como parâmetros em operações de entrada e saída.
Binding	Define o protocolo e formato que serão utilizados para fornecer o serviço.
Service	Especifica um ou mais *endpoints*[1] nos quais a funcionalidade do serviço está disponível.

[1] Um *endpoint* é uma associação entre um *binding* e um endereço de rede, especificado por uma URI. Um *endpoint* indica um local específico para acessar um serviço usando um protocolo específico e formato de dados definido. [http://www.w3.org/TR/ws-gloss/]

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://uniriotec.monografia.br/HelloWorld/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="HelloWorld"
    targetNamespace="http://uniriotec.monografia.br/HelloWorld/">
    <wsdl:types>
        <xsd:schema targetNamespace="http://uniriotec.monografia.br/HelloWorld/">
            <xsd:element name="sayHelloRequest">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="in" type="xsd:string" />
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="sayHelloResponse">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="out" type="xsd:string" />
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:schema>
    </wsdl:types>
    <wsdl:message name="sayHelloRequest">
        <wsdl:part element="tns:sayHelloRequest" name="fullName" />
    </wsdl:message>
    <wsdl:message name="sayHelloResponse">
        <wsdl:part element="tns:sayHelloResponse" name="greeting" />
    </wsdl:message>
    <wsdl:portType name="HelloWorld">
        <wsdl:operation name="sayHello">
            <wsdl:input message="tns:sayHelloRequest" />
            <wsdl:output message="tns:sayHelloResponse" />
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="HelloWorldSOAP" type="tns:HelloWorld">
        <soap:binding style="document"
            transport="http://schemas.xmlsoap.org/soap/http" />
        <wsdl:operation name="sayHello">
            <soap:operation soapAction="http://uniriotec.monografia.br/HelloWorld/sayHello" />
            <wsdl:input>
                <soap:body use="literal" />
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal" />
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="HelloWorld">
        <wsdl:port binding="tns:HelloWorldSOAP" name="HelloWorldSOAP">
            <soap:address location="http://localhost:8080/hello" />
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Figura 2 - Exemplo de arquivo WSDL na vesão 1.1

2.4 SOAP

Conforme visto na seção 2.3, o WSDL descreve a parte do contrato de serviço responsável pela invocação do web service. As operações contidas no WSDL são descritas em termos de mensagens abstratas e o Protocolo de Acesso Simples a Objeto (SOAP – Simple Object Access Protocol) é a especificação utilizada em web services para fornecer implementação concreta para as mensagens definidas em

WSDL 1.1 (versão utilizada neste trabalho), definindo a estrutura XML das mensagens trocadas de forma padronizada.

SOAP inclui um modelo de processamento composto por nós SOAP que transmitem e recebem mensagens SOAP e pode retransmiti-las para outros nós SOAP [Hansen, 2007]. Uma mensagem SOAP, tanto de requisição quanto de resposta, é dividida em três partes:

- ***Envelope*:** define o início e fim de uma mensagem SOAP e os *namespaces* utilizados na mensagem. A Figura 3 mostra uma mensagem SOAP de requisição a um serviço e uma possível mensagem de resposta a esta requisição. Nesta mensagem, é apresentado o elemento *Envelope* com prefixo soapenv e a declaração dos *namespaces* que definem os elementos utilizados pela versão SOAP 1.1 (http://schemas.xmlsoap.org/soap/envelope/) e as mensagens definidas no serviço *TratarProduto* referentes ao método *consultarProduto*.
- ***Header***: contém informações relacionadas à qualidade de serviço (QoS) e informações relacionadas à segurança e confiabilidade. *Headers* não são obrigatórios e não foram configurados no exemplo apresentado na Figura 3.
- ***Body*:** tanto as mensagens de requisição quanto as mensagens de resposta devem ter apenas um elemento filho na seção *Body*. Esta abordagem é coerente com a definição de mensagens no WSDL, já que para cada operação definida é utilizada apenas uma mensagem para a requisição e uma mensagem para a resposta. A Figura 3, mostra que o elemento *Body* da requisição contém um único filho chamado “consutarProduto”. Esse elemento contém o tipo simples “codigo”. Da mesma forma, a mensagem de resposta da Figura 3, mostra que seu elemento *Body* também contém apenas um filho chamado “consultarProdutoResponse”. Esse elemento contém o tipo complexo “produto” que é composto pelos tipos simples “codigo”, “nome” e “valor”.

Requisição

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:trat="http://www.uniriotec.monografia.br/TratarProduto/">
   <soapenv:Header/>
   <soapenv:Body>
      <trat:consultarProduto>
         <codigo>12345</codigo>
      </trat:consultarProduto>
   </soapenv:Body>
</soapenv:Envelope>
```

Resposta

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:trat="http://www.uniriotec.monografia.br/TratarProduto"
                  xmlns="http://uniriotec.monografia.br/ModeloCanonico">
   <soapenv:Header/>
   <soapenv:Body>
      <trat:consultarProdutoResponse>
         <produto>
            <codigo>126327</codigo>
            <nome>notebook</nome>
            <valor>1450,00</valor>
         </produto>
      </trat:consultarProdutoResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

Figura 3 - Exemplo de mensagem SOAP

2.5 XSD

XML Schema Definition (XSD) é a linguagem de definição de esquemas recomendada pela W3C, escrita em linguagem XML 1.0 e utilizada para descrição de estrutura de instância de documentos [Wyke e Watt, 2002]. A partir da definição destas estruturas torna-se possível trocar informações entre aplicações com maior confiabilidade. Uma das utilizações de um XSD é a definição de estruturas de dados de um WSDL.

Os principais elementos de um XSD são *symple type definitions*, *complex type definitions*, *element declarations* e *atribute declarations*. A Tabela 2 apresenta os principais elementos de um XSD 1.0, versão utilizada neste trabalho.

Tabela 2 - Elementos de um XSD

Elemento	Descrição
Import	Define um outro XSD para ser importado e utilizado ou estendido.
simple type	Define um tipo de dado simples que pode ser reutilizado por um elemento ou em uma mensagem

complex type	Define um tipo complexo de dados para ser reutilizado por um elemento ou mensagem. Tipos complexos são estruturas semelhantes a classes Java.
elemento	Define um elemento para ser utilizado em uma mensagem. Um elemento pode fazer parte de um tipo simples ou complexo.
atribute	Define uma característica para um elemento ou tipo de dados. Podem ser utilizados para definir restrições.
documentation	Utilizado para documentação do esquema XML, semelhante à seção *documentation* de um documento WSDL.

A Figura 4 apresenta um exemplo de esquema com os elementos principais para transferir dados cadastrais de uma pessoa. Neste exemplo, é possível observar, na representação do tipo complexo Pessoa, a utilização dos elementos representando suas informações estruturadas em: nome completo, cpf, endereço, telefone e renda. Além disso, o atributo endereço corresponde ao uso da estrutura referente ao tipo complexo endereço (definida como contendo os elementos rua, número, bairro, cidade e estado). Uma mensagem SOAP poderia tanto utilizar o elemento raiz "pessoa", ou utilizar diretamente o tipo complexo Pessoa.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.uniriotec.br/Pessoa"
    xmlns:tns="http://www.uniriotec.br/Pessoa" elementFormDefault="qualified">

    <annotation>
        <documentation>
            Este esquema define a estrutura necessária para troca de dados de pessoa
            física.
    </documentation>
    </annotation>

    <element name="pessoa" type="tns:Pessoa" />

    <complexType name="Endereco">
        <sequence>
            <element name="rua" type="string" nillable="false" />
            <element name="numero" type="integer" />
            <element name="bairro" type="string" />
            <element name="cidade" type="string" />
            <element name="estado" type="string" />
        </sequence>
        <attribute name="pais" type="NMTOKEN" fixed="BR" />
    </complexType>

    <complexType name="Pessoa">
        <sequence>
            <element name="nomeCompleto" type="string" nillable="false" />
            <element name="cpf" type="integer" nillable="false" />
            <element name="endereco" type="tns:Endereco" maxOccurs="1" />
            <element name="telefone" type="integer" />
            <element name="renda" type="double" />
        </sequence>
    </complexType>
</schema>
```

Figura 4 - Exemplo de um XML Schema Definition

Além dos elementos já apresentados, também é possível perceber a utilização de restrições de valores pré-definidos, obrigatoriedade de preenchimento, quantidade de valores máxima, entre outros. Estas restrições adicionam ao esquema a possibilidade de realizar validações antes mesmo que uma mensagem seja enviada a um serviço. No exemplo apresentado, uma mensagem com o cpf de uma pessoa não preenchido ou a existência de mais de um endereço tornaria esta mensagem inválida antes mesmo da mensagem chegar ao provedor do serviço. A Tabela 3 apresenta exemplos de restrições que podem ser especificadas em um XSD.

Tabela 3 - Lista de Restrições em um XSD

Restrição	Descrição
enumeration	Define uma lista de valores válidos.
fractionDigits	Especifica o número máximo de casas decimais permitidas.
Length	Especifica o número exato de caracteres ou itens permitidos.
maxExclusive	Especifica o valor máximo para valores numéricos.
maxInclusive	Especifica o valor máximo para valores numéricos.
maxLength	Especifica o número máximo de caracteres ou itens permitidos.
minExclusive	Especifica o valor mínimo (exclusive) para valores numéricos, ou seja, o valor do campo deve ser maior que ao valor definido.
minInclusive	Especifica o valor mínimo (inclusive) para valores numéricos, ou seja, o valor deve ser maior ou igual ao valor definido.
minLength	Especifica o número mínimo de caracteres ou itens permitidos
Pattern	Define a sequência exata de caracteres permitidos.
totalDigits	Especifica o número exato de dígitos permitidos.
whiteSpace	Especifica como caracteres vazios (tabs, espaços e retornos de carro) são tratados.

Em SOA, trocas de informações entre serviços e/ou aplicações distribuídas são constantes e fundamentais. As mensagens trocadas possuem informações que devem estar estruturadas e especificadas em um contrato, podendo ser definidas em um XSD. ESB

Um dos principais componentes de SOA é o ESB (Enterprise Service Bus) que é a infraestrutura de mediação que permite invocar serviços em um ambiente de sistemas distribuídos sem ser conexão ponto-a-ponto [Hewitt, 2009]. O ESB deve ser capaz de fornecer um meio de comunicação entre provedor e consumidor que seja estável, seguro, simples, integrado e independente de plataforma. A seguir, algumas responsabilidades atribuídas ao ESB:

- **Prover conectividade**

Na conexão ponto a ponto, a solicitação de requisição do serviço é feita utilizando o *endpoint* onde o serviço está publicado. Neste tipo de abordagem, se o endereço físico onde o serviço está alocado for alterado, o consumidor não poderá acessá-lo enquanto não atualizar o endereço do serviço. Na conexão por mediação, o consumidor não conhece o endereço físico do serviço. A requisição é feita ao ESB que identifica o *endpoint* apropriado para o serviço. Essa abordagem possibilita maior gerenciamento sobre mudanças, possibilidade do mesmo serviço ser provido em diferentes sistemas e possibilita balanceamento de carga.

- **Transformação de dados**

Como a requisição do serviço é recebida pelo ESB e este é o responsável pela integração entre consumidor e provedor, ele deve ser capaz de fazer transformações nos dados da requisição para o formato entendido pelo consumidor e pelo serviço provido. Em um ESB, os dados podem ser transformados utilizando XSLT e consultados utilizando XQuery e XPath [Sousa *et al.*, 2010], garantindo conectividade e interoperabilidade entre consumidores e provedores que utilizam tipos de dados distintos.

- **Roteamento inteligente**

O roteamento inteligente possibilita que mensagens de consumidores possam ser encaminhadas a provedores específicos baseado em regras de negócios, transformação de dados e adaptadores para aplicações [Papazoglou, 2007].

- **Tratar segurança**

O ESB deve prover segurança aos consumidores dos serviços como criptografia, autenticação, validação de cada solicitação de serviço e autorização para validar os privilégios dos consumidores quanto ao acesso do serviço e codificação/ decodificação de conteúdo XML [Papazoglou, 2007].

- **Administração de serviços**

Manter repositório e registro de serviços disponíveis, permitindo que tanto consumidor ou provedor possam identificar os serviços e utilizá-los.

- **Monitoramento e *logging***

Dado que o ESB é o mediador de troca de mensagens entre aplicações distribuídas, o ESB pode gerar arquivos de *log*, tanto de erros quanto de informações. Isto permite monitorar o funcionamento das aplicações.

A Figura 5 mostra um exemplo de como o ESB provê conectividade e interoperabilidade entre consumidores e provedores de serviços desenvolvidos em plataformas distintas. O barramento fornece, por exemplo, abstração do destino físico e as informações de conexão (como número de TCP / *hostname* e porta IP) permitindo a conexão aos serviços utilizando nomes de conexão lógica que são mapeados pelo ESB para saber o destino real de cada serviço. Este desacoplamento lógico oferece aos serviços publicados no ESB a capacidade de serem atualizados, movidos ou substituídos sem gerar impacto no consumidor do serviço [Papazoglou, 2007].

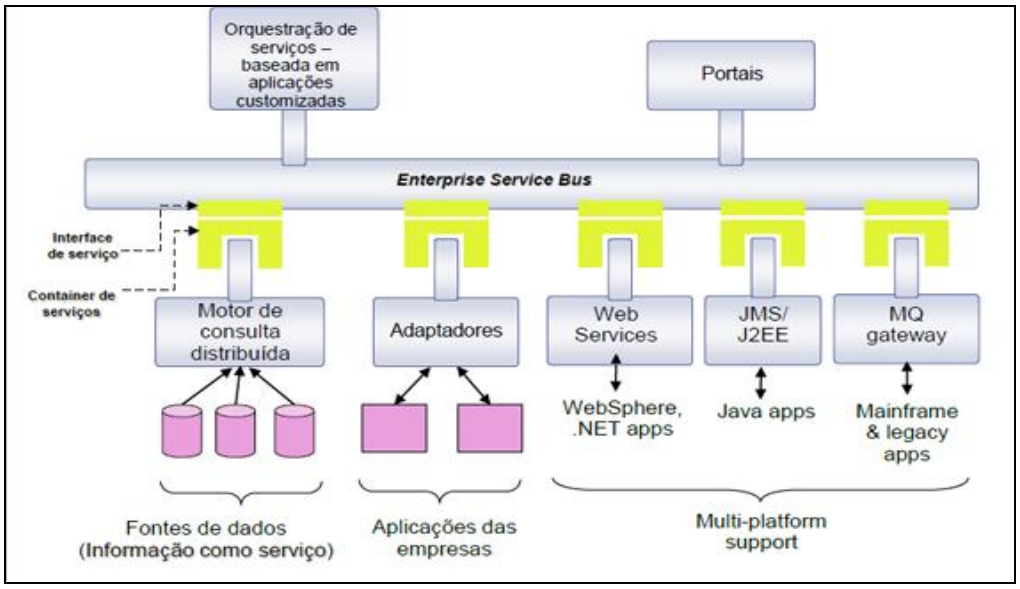

Figura 5 - Enterprise Service Bus (adaptado de [Papazoglou, 2007]).

2.6 BPM

A Gestão de Processos de Negócios (Business Process Management - BPM) se refere a todas as atividades que visam gerenciar os processos de negócio de forma sistemática. Essas atividades normalmente são apoiadas por meios tecnológicos e

visam trazer visibilidade e controle sobre os processos para que seus gestores sejam capazes de identificar oportunidade de melhorias [Josuttis, 2007].

Processos bem definidos como Modelagem de Processos de Negócio (Business Process Modeling), Análise de Processos de Negócio (Business Process Analisys), Melhoria de Processos de Negócio (Business Process Improvement - BPI) e Integração de Processos de Negócio (Business Process Integration - BPI) fazem parte desse conceito que é a Gestão de Processos de Negócios.

Em particular, a modelagem de processos de negócio compreende a construção de um conjunto de visões integradas que provê entendimento comum do negócio e serve como base para comunicação, melhoria e inovação [Magalhães 2007]. A Figura 6 apresenta as principais perspectivas envolvendo os conceitos contemplados em um modelo de processos de negócio e as descreve a seguir.

- **Modelo organizacional (Quem?):** documenta a hierarquia da organização (áreas, grupos e papéis) permitindo sua visualização do nível mais alto até os papéis organizacionais;
- **Modelo de localização (Onde?):** apresenta de forma hierárquica as localizações geográficas da organização;
- **Modelo de processos (Como?):** apresenta a sequência de atividades que compõem um processo de negócio;
- **Modelo de atividades (Como?):** apresenta as informações necessárias de uma atividade, ou seja, o que é necessário para sua execução como: informações de entrada e saída, sistemas que apoiam a atividade, papel que executa a atividade, entre outros;
- **Modelo de objetivos (Porquê):** apresenta os objetivos de negócio da organização de forma hierárquica;
- **Modelo de eventos (Quando?):** estabelece o momento de execução de uma determinada atividade. Os eventos podem ser temporais, externos à organização, provocados pela execução de outra atividade ou por seu término;

- **Modelo de produtos (O quê?):** apresenta um resumo dos artefatos gerados por um processo;
- **Modelo de sistemas (O quê?):** representa os sistemas que apóiam a execução de atividades do processo de negócio.

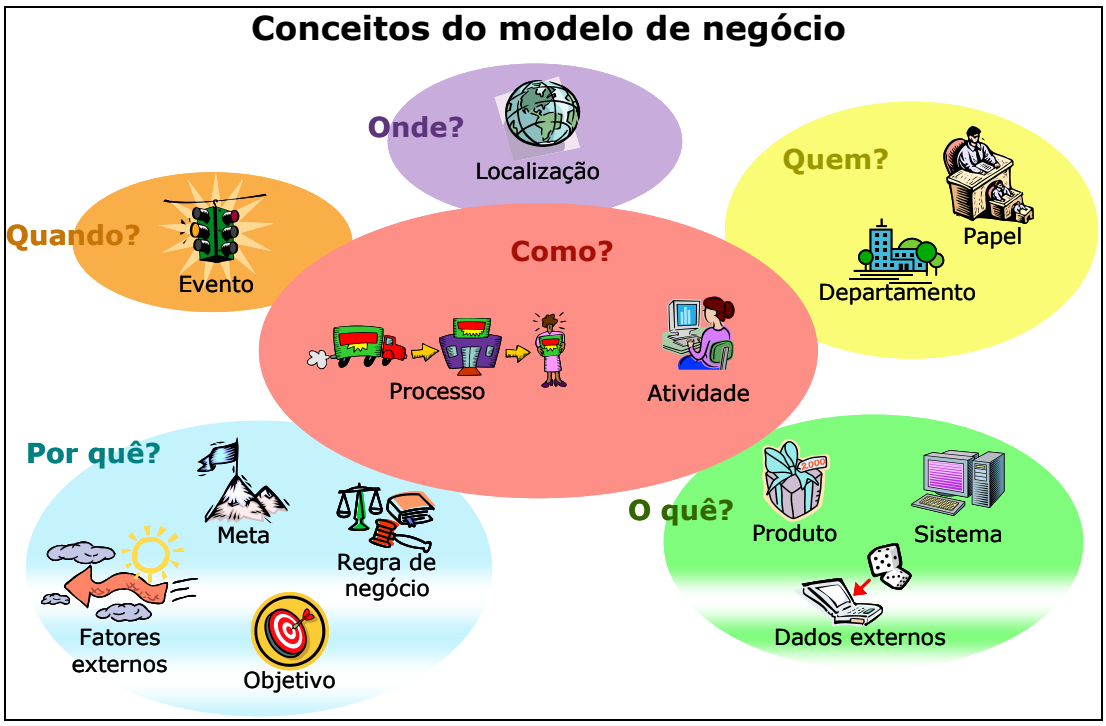

Figura 6 - Perspectivas dos modelos de negócios [Magalhães, 2007].

Entre os diferentes modelos que compõem o modelo de negócio, destaca-se o modelo de processos, que estabelece a sequência de atividades que compõem um processo. Essencialmente, um processo é composto por um conjunto de atividades. Estas atividades são partes bem caracterizadas do trabalho, realizadas em certo momento por papéis, de acordo com um conjunto de regras definidas que estabelecem a ordem e as condições em que as atividades devem ser executadas. Cada atividade manipula um conjunto de produtos de trabalho (dados, documentos ou formulários) durante sua execução [Sharp e McDermott, 2001].

A implantação de SOA em uma organização não é algo trivial; ao contrário, requer esforços que vão além do escopo técnico, como necessidade de alinhamento entre o negócio e a TI e mudança na cultura organizacional [Marks e Bell 2006]. Empresas com uma cultura orientada a processos conseguem entender melhor a necessidade de romper as fronteiras da verticalização departamental e enxergar a

instituição de forma horizontal, entendendo que a produção de um produto ou serviço ultrapassa as fronteiras departamentais e, muitas vezes, só faz sentido através da interação/integração com pessoas e processos de outros departamentos.

A afirmativa acima não significa que empresas sem a cultura de gerenciamento de seus processos estão fadadas a não conseguir implantar um modelo SOA, mas certamente esse será mais um entre tantos fatores complicadores. Serviços apresentados em SOA são parte dos processos de negócio das empresas. Logo, se estes processos são bem definidos, conhecidos e gerenciados, o que se deve fazer é identificá-los.

Através da modelagem de processos de negócio, é possível ter uma visão sistemática das responsabilidades de cada papel, das informações necessárias e produzidas em cada atividade do processo, dos artefatos gerados, da sequência de atividades, das regras de negócio e dos sistemas que apoiam o processo.

Azevedo *et al.* [2009a, 2009b, 2009c, 2009e] apresentam heurísticas para identificação de serviços candidatos que utilizam exatamente as informações provenientes de processos de negócio modelados, conforme será abordado sucintamente na seção 2.8.

2.7 Ciclo de Vida de Serviços

Gu e Lago [2007] argumentam que em SOA existem novas e diferentes tarefas e adição de papéis que não pertencem ao modelo tradicional de engenharia de software para desenvolvimento de serviços. Para alcançar alguns objetivos de SOA, como reuso, interoperabilidade e durabilidade, um ciclo de vida sistemático e bem definido é crucial. Além disso, considerando as características de SOA, surgem novos desafios, tais como requisitos conflitantes, tratamento de segurança considerando serviços distribuídos e alinhamento das soluções entre negócio e TI. Gu e Lago [2007] avaliaram alguns modelos de ciclo de vida da literatura e da indústria e propuseram um novo modelo de ciclo de vida evoluindo as propostas existentes baseado em papéis (ou *stakeholder*), englobando as características referentes à construção e à manutenção de serviços. No molde apresentado, estão abordadas não somente as etapas de desenvolvimento do provedor, como também a

visão do consumidor e do *broker*. O objetivo da proposta é obter uma melhor gerência sobre serviços para prover serviços de alta qualidade.

Neste trabalho, o foco está nas atividades desempenhadas pelo provedor de serviços. Portanto, as atividades de responsabilidade do broker e do consumidor dos serviços não foram tratadas ficando como trabalhos futuros considerando a extensão do presente trabalho.

A seguir são descritas as atividades do provedor de serviços considerando as três fases utilizadas por Gu e Lago [2007] na definição do ciclo de vida: Projeto, Execução e Mudança.

2.7.1 Fase de Projeto

Esta fase engloba os passos entre a identificação e descoberta do serviço e o desenvolvimento e teste do mesmo, que podem ser descritos da seguinte forma.

- *Pesquisa de Mercado* – O objetivo deste passo é avaliar o mercado e orientar o desenvolvimento do serviço. Uma boa pesquisa previne a criação de serviços não benéficos ou com pouco uso;

- *Engenharia de Requisitos* – Passo onde as ideias e demandas são coletadas. São avaliados alguns pontos como: potencial de reusabilidade, maior adequação às necessidades de mercado, acesso ao serviço pelos consumidores e manutenibilidade do serviço;

- *Modelagem de Negócio* – Após o levantamento de requisitos, o objetivo é modelar o processo de negócio referente ao serviço, definindo escopo e objetivo do serviço. Normalmente, um analista de negócio deve participar dessa etapa;

- *Projeto do Serviço* – Nesta fase, o arquiteto define o projeto do serviço, definindo os requisitos funcionais e não-funcionais. A parte técnica do contrato é definida nesta fase. Além disso, é feita pesquisa no registro de serviços para identificar aqueles que podem ser reutilizados;

- *Desenvolvimento do Serviço* – A equipe de desenvolvimento implementa o serviço especificado de acordo com os requisitos definidos. Além disso, testes de integração com outros serviços devem ser realizados bem como e testes do serviço em si. Somente após estar aderente aos requisitos, o serviço é liberado para a equipe de testes;

- *Testes do serviço* – Passo onde deve ser garantida a liberação do serviço para uso comercial. O objetivo não é somente identificar falhas, mas garantir a qualidade do serviço.

2.7.2 Fase de Execução

Esta fase engloba os passos de publicação, disponibilização e monitoramento do serviço registrado, os quais são descritos a seguir.

- *Publicação do Serviço* – Após obter um serviço desenvolvido e testado, esta atividade tem o objetivo de publicá-lo em um registro de serviços, a fim de que o serviço seja mais facilmente localizado pelos seus possíveis consumidores;

- *Disponibilização do Serviço* – Nesta etapa, há uma interação com o consumidor para definir um acordo de nível de serviço. No contrato, podem ser consideradas informações de custo, disponibilidade, desempenho do serviço e regras de acesso. Estas características definem o que é chamado de nível de acordo de serviço (SLA – Service Level Agreement), o qual define um conjunto de acordos entre consumidor e provedor. Após a assinatura da SLA, o consumidor obtém autorização e acesso ao serviço;

- *Monitoramento do Serviço* – Este passo consiste em monitorar o estado do serviço considerando, dentre outras coisas, tempo de resposta, falhas durante a execução, disponibilidade, etc. Enquanto o serviço estiver sendo disponibilizado, esta etapa se mantém ativa, para garantir que os termos da SLA sejam atendidos.

2.7.3 Fase de Mudança

Esta fase representa o processo de manutenção e mudança do serviço. Nesta fase, são avaliadas possíveis mudanças nos requisitos de negócio ou nos níveis de acordo de serviço definidos. Caso sejam necessárias alterações, o ciclo de vida do serviço deve retornar à engenharia de requisitos (ou outra etapa anterior de acordo com a necessidade) e passar novamente pelas etapas seguintes no ciclo de vida.

2.8 Métodos para Identificação, Análise e Projeto de Serviços

Azevedo *et al.* [2009e] ajustaram a proposta de Gu e Lago [2007] para contemplar um ciclo de vida de desenvolvimento de serviços em uma organização orientada a gestão de processos de negócio, ou seja, que tenha iniciativas de modelagem de processos de negócio. Esta é uma premissa deste trabalho.

A proposta de Azevedo *et al.*[2009e] inclui a substituição da atividade "Mapeamento do mercado" por "Modelagem do negócio". Esta atividade tem como objetivo ter a modelagem dos processos de negócio da organização, a fim de que estes sirvam como base para a identificação de serviços, a partir de uma demanda de desenvolvimento de sistema. Além disso, foi proposta a inclusão de uma atividade de identificação de serviços a partir de processos de negócio.

Neste trabalho, é considerado que os processos de negócio referentes à demanda de implementação de serviços já estão elaborados. Logo, os passos tratados neste trabalho iniciam a partir da identificação de serviços.

Esta seção apresenta resumidamente os métodos para identificação, análise e projeto de serviços a partir da modelagem de processos. Maiores detalhes podem ser encontrados em Azevedo *et al.* [2009a, 2009b, 2009c, 2009e, 2001] e Dirr *et al.* [2012].

2.8.1 Identificação de Serviços

A identificação de serviços efetua uma análise *top-down* dos modelos de processos, a fim de identificar **serviços candidatos** a partir de elementos sintáticos e semânticos de modelos de processos de negócio. Um serviço candidato é uma abstração (não implementada) de serviço identificada durante a fase de identificação de serviços que poderá ser implementada futuramente como serviço ou como funcionalidade de uma aplicação [Erl, 2005]. Os serviços candidatos identificados são classificados em:

- Serviço candidato de dados: Executa apenas operações de acesso a dados, tais como, criar, recuperar, atualizar e remover, ou seja, operações CRUD (*Create*, *Retrive*, *Update* e *Delete*);
- Serviço candidato de lógica: Serviço que executa uma lógica do negócio. Devido à abstração na definição do serviço, estes serviços não estão apenas limitados à execução de lógica de negócio, podendo também executar operações CRUD. No entanto, o foco desta classe de serviços está em executar lógica do negócio;
- Serviço candidato utilitário: Serviço que provê funções cuja estrutura pode ser generalizada e utilizada em mais de um processo de negócio.

Cada serviço candidato encontrado é descrito considerando um conjunto de informações, as quais são apresentadas na Tabela 4.

Tabela 4 - Template para descrição de serviço candidato

Nome	<nome do serviço candidato>
Tipo	<indicação do tipo de serviço candidato: negócio, dado, utilitário>
Entrada	<informação necessária para o serviço poder executar>
Saída	<informação retornada pelo serviço>
Origem	<indicação da heurística utilizada para identificar o serviço>
Atividades	<lista de atividades consideradas para identificação deste serviço candidato>
Padrão recorrente	<quando constatado para o serviço candidato, este campo indica o padrão baseado em função recorrente em fluxos de processos de negócio ou padrão relacionado a aspectos específicos de estrutura organizacional>
Descrição	<breve descrição do serviço candidato>
Observação	<informações relevantes para a fase de projeto de serviços>

Por ser um cenário bastante variado, com grande número possível de soluções, a alternativa encontrada para a identificação de serviços candidatos foi o uso de métodos heurísticos. A seguir são listadas as heurísticas do método de identificação de serviços. Maiores detalhes são apresentados em Azevedo *et al.* [2009a, 2009b, 2009c, 2009e].

- Heurística de regras de negócio - *Toda regra de negócio deve ser identificada como um serviço candidato;*
- Heurística de requisitos de negócio – *Todo requisito de negócio deve ser identificado como um serviço candidato;*

- Heurística de informações entrada/saída – *Toda informação de entrada e saída de uma atividade deve ser identificada como um serviço candidato de leitura/escrita em uma base de dados;*
- Heurística de atividades seqüenciais – *Para um conjunto de atividades seqüenciais automatizadas ou apoiadas por sistema, deve ser identificado um serviço candidato;*
- Heurística de AND – *Um serviço candidato deve ser identificado a partir de uma estrutura iniciada em um ponto do workflow, na qual o fluxo subdivide-se em processos executados em paralelo, e finalizada em um ponto onde esses processos convergem novamente;*
- Heurística de XOR – *Um serviço candidato deve ser identificado a partir de uma estrutura iniciada em um ponto no workflow onde, baseado em uma decisão, uma e somente uma de várias ramificações do fluxo é escolhida, e finalizada em um ponto no workflow onde as ramificações do fluxo se juntem sem sincronização ou quando uma ou mais das ramificações termine em evento final;*
- Heurística de OR – *Um serviço candidato deve ser identificado a partir de uma estrutura iniciada em um ponto no workflow onde, baseado em uma decisão, uma ou mais ramificações do fluxo são escolhidas e finalizadas em um ponto no workflow onde as várias ramificações do fluxo se juntem. Se mais de um dos fluxos de origem for executado, então é necessário sincronizá-los. Ramificações podem também terminar em um evento final;*
- Heurística de Loop – *Um serviço candidato deve ser identificado a partir de uma estrutura do workflow onde uma ou mais atividades podem ser executadas repetidamente, formando um ciclo;*
- Heurística de interface de processo – *Caso um processo seja iniciado por uma interface e seja seguido de uma ou mais atividades automatizadas, um serviço candidato deve ser identificado. Da mesma forma, caso um processo seja finalizado por uma interface e a atividade(s) anterior(es) seja automatizada, um serviço candidato deve ser identificado;*

- Heurística de atividade de múltiplas instâncias – *Toda atividade de múltiplas instâncias deve dar origem aos seguintes serviços candidatos: (i) serviço candidato para o remetente da mensagem transmitir as mensagems para os destinatários; (ii) serviço candidato para cada destinatário; e (iii) serviço candidato para consolidar as execuções de diferentes instâncias e repassá-las para o remetente.*

Após a execução de cada uma das heurísticas, tem-se uma lista de serviços candidatos resultante. Entretanto, existe a possibilidade de termos serviços duplicados, serviços que na realidade não precisarão ser implementados, ou mesmo serviços que estão com uma granularidade muito fina para serem implementados como serviços separados. Dessa forma, é necessário levantar informações dos serviços a fim de auxiliar o analista SOA a analisar os serviços gerados. Isto é feito no passo de consolidação de serviços candidatos. Este passo tem o objetivo de gerar um conjunto de informações em tabelas, gráficos, grafos de dependência, etc. que serão utilizadas por analistas SOA para decidir como os serviços candidatos serão implementados. Esta etapa inclui a execução das seguintes heurísticas.

- Heurística de eliminação de serviços candidatos - *Devem ser eliminados todos os serviços candidatos que apareçam duplicados ou que refiram a regras que explicitam valores padrão;*

- Heurística de grau de reuso de serviço candidato - *O grau de reuso de um serviço candidato identificado a partir de uma atividade corresponde à soma do número de ocorrências de cada atividade em que o serviço está relacionado. O grau de reuso de um serviço candidato identificado a partir de um conjunto de atividades é igual ao número de ocorrências do conjunto de atividades na mesma ordem que esta foi definida no serviço. Além disso, quando um serviço candidato A faz parte da composição de outro serviço candidato B, então deve-se somar o grau de reuso do serviço composto B ao grau de reuso do serviço A calculado previamente;*

- Heurística de grau de reuso de serviço candidato identificado a partir de atividade de múltiplas instâncias - *O serviço identificado a partir de atividades de múltiplas instâncias deve conter uma marcação especial a fim*

de indicar que o seu reuso é maior do que o calculado pela heurística de grau de reuso de serviço candidato, pois ele também é reutilizado por cada instância da atividade, isto é, seu uso é, em geral, maior do que serviços não identificados a partir de atividades de múltiplas instâncias;

- Heurística de associação de serviços candidatos com sistemas - *Um serviço candidato identificado a partir de um requisito de negócio deve ser associado aos sistemas que o implementam o requisito, caso ele já tenha sido implementado;*
- Heurística de associação de serviços candidatos com requisitos da demanda - *Um serviço candidato identificado a partir de um requisito de negócio da demanda deve ser associado ao respectivo requisito da demanda;*
- Heurística de associação de serviços com papéis - *Um serviço candidato deve ser associado aos papéis que executam a(s) atividade(s) que lhe deram origem;*
- Heurística de associação de serviços com atividades - *Um serviço candidato deve ser associado à(s) atividade(s) que lhe deram origem;*
- Heurística de associação de serviço de dados - *Serviços de dados de leitura ou escrita identificados a partir da mesma informação de entrada/saída devem ser associados um ao outro;*
- Heurística de associação de clusters com modelo conceitual - *Serviços de dados identificados a partir de informações que representem a mesma visão dos dados da organização devem ser associados uns aos outros;*
- Heurística de associação de serviços candidatos a partir de associações entre elementos do modelo - *Um serviço candidato deve ser associado aos serviços candidatos que o utilizam. Esta informação é obtida a partir da associação entre regra de negócio × regra de negócio, regra de negócio × requisito de negócio, regra de negócio × cluster e requisito de negócio × cluster, quando esta está explícita na modelagem;*
- Heurística de identificação de serviços utilitários - *Um serviço utilitário candidato deve ser identificado a partir de um ou mais serviços candidatos*

previamente identificados quando, pela análise dos modelos de processos de negócio, for constatado que este(s) serviço(s) faz(em) parte de um padrão baseado em funções recorrentes de processos de negócio ou de um padrão relacionado a aspectos específicos de estrutura organizacional que podem ser generalizados [Thom et al.,2007].

Ao fim desta etapa, têm-se uma lista de serviços candidatos com informações (presentes, por exemplo, em tabelas, grafos e gráficos) que ajudarão o analista a decidir quais devem ser implementados como serviços de software (por exemplo, como *web services*) ou serão implementados como funcionalidades de aplicações. A etapa seguinte ajuda a tomar esta decisão, e corresponde à análise de serviços.

2.8.2 Análise de Serviços

Nesta fase é realizada a análise dos serviços candidatos identificados anteriormente com objetivo de produzir um grupo de serviços com nível de granularidade apropriado para a fase de projeto. Os serviços resultantes devem estar alinhados aos requisitos de negócio e, ao mesmo tempo, deve haver preocupações como durabilidade e reuso.

Para auxiliar nesta tomada de decisão são definidos os seguintes passos: (i) Priorização de serviços candidatos; e (ii) Definição de grupos de serviços. Detalhes desta fase são apresentadas por Azevedo *et al.* (2011) e descritos de forma resumida a seguir.

2.8.2.1 Priorizar serviços candidatos

Nesta etapa os serviços de maior importância e maior contribuição para a organização serão destacados através do cálculo da prioridade de cada serviço candidato. As heurísticas utilizadas nesta etapa são descritas a seguir.

- Heurística de grau de reuso de cada serviço candidato - *Agrupar serviços candidatos para cada diferente grau de reuso encontrado, listando grau de reuso e quantidade de serviços candidatos correspondente ao grau de reuso. Em seguida, definir pesos para cada agrupamento;*

- Heurística de associação de serviços candidatos com sistemas - *Agrupar serviços candidatos que estão associados a cada sistema que os implementa, listando sistema e quantidade de serviços candidatos. Em seguida, definir pesos para cada agrupamento;*
- Heurística de aumento de peso de serviços candidatos identificados a partir de atividades de múltiplas instâncias - *Atribuir maior peso para serviços candidatos identificados a partir de atividades de múltiplas instâncias;*
- Heurística de associação de serviços candidatos com requisitos da demanda - *Definir pesos para serviços candidatos associados aos requisitos da demanda. Definir maior peso para serviços candidatos associados aos requisitos de uma demanda com maior prioridade;*
- Heurística de serviços candidatos com papéis - *Agrupar serviços candidatos que estão associados a cada papel, listando papel e quantidade de serviços candidatos. Em seguida, definir pesos para cada agrupamento;*
- Heurística de cálculo de priorização de serviços candidatos - *Calcular priorização de serviços candidatos de acordo com o somatório dos pesos definidos em cada heurística de priorização;*
- Heurística de detalhamento de priorização de serviços candidatos a partir de fluxo - *Priorizar serviços de fluxo de acordo com as informações levantadas para caracterização dos mesmos, tais como: número de atividades que compõem o fluxo; número de modelos nos quais o mesmo fluxo é reutilizado; número de entidades tratadas pelo fluxo; e número de raias diferentes existentes no fluxo do serviço; número de sub-fluxos. Para cada uma destas informações, pesos devem ser definidos, atribuídos a cada caso e o somatório dos pesos corresponde à importância dos serviços de fluxo;*

- Heurística de granularidade de serviços candidatos – *Elaborar um mapa de granularidade e posicionar serviços com menor granularidade (grossos) no topo enquanto que serviços com maior granularidade (finos) ficam na base do mapa. Serviços com granularidades intermediárias ficam entre os serviços grossos e finos de acordo com sua granularidade.*

2.8.2.2 Definição de grupos de serviços

Nesta fase, o objetivo é agrupar os serviços candidatos, sejam de dados ou de lógica, de acordo com similaridade de operações ou por manipularem o mesmo grupo de entidades. Deve ser considerado o reuso dos serviços para que o agrupamento seja feito adequadamente. As heurísticas utilizadas neste passo são resumidas a seguir.

- Heurística de agrupamento de serviços de dados – Associar serviços de dados de acordo com entidades e operações CRUD. Definir canais de comunicação para operações de leitura e reduzir custo de comunicação entre os mesmos;
- Heurística de agrupamento de serviços de lógica – Agrupar serviços de lógica de acordo com os padrões de uso do serviço na organização.

2.8.3 Projeto de Serviços

Aplicações SOA consistem principalmente pela composição de serviços interoperáveis, padronizados e reutilizáveis em outras composições ou aplicações [Erl, 2007]. Para que estes objetivos sejam alcançados, após a análise dos serviços candidatos, eles devem ser projetados considerando princípios de projeto, tais como: padronização de contrato de serviço, utilização de serviços existentes, composição, abstração de serviços, baixo acoplamento, reuso e autonomia [Erl, 2008].

Algumas heurísticas podem ser utilizadas para que o projeto possa seguir os padrões abordados. Tais heurísticas são apresentadas por [Azevedo *et al.*, 2009c] e resumidas na Tabela 5.

Tabela 5 - Resumo de projeto de serviços [Azevedo *et al.*, 2009c]

Princípio	Descrição
Comunicação assíncrona	Para cada serviço, priorizar implementação utilizando comunicação assíncrona
Tipo de dados heterogêneos	Antes de criar um tipo de dado na implementação do serviço, verificar os tipos de dados existentes no modelo canônico.
	Não definir tipos de dados homogêneos para todos os contratos dos serviços.
Mediadores	Utilizar mediadores para invocação de serviços.
Verificação de tipos	Utilizar verificação de tipos nos mediadores quando o contrato do serviço estiver estável.
Padrão de interação	Utilizar tipos de dados comuns em diversas linguagens de programação.
Controle de transação	Definir operações de compensação para as operações do serviço.
Controle da lógica do processo	Em ambientes controlados, onde se ter um controlador central não causa problemas de gargalo, utilizar orquestração de serviços para composição de serviços. Caso o controlador central esteja impactando o desempenho da composição, utilize coreografia de serviços.
Versionamento	Utilizar transformação de mensagens no barramento para realizar um versionamento implícito dos tipos de dados.
Especificação	Separar a especificação do serviço da sua implementação, descrevendo um modelo técnico (contrato) e um modelo conceitual (requisitos, restrições, etc.).
	Descrever formalmente a especificação do serviço.
Reuso	Prover nomes significativos e bem descritivos do serviço e suas operações baseado no domínio do negócio, não na tecnologia.
	Publicar descrição dos serviços em um repositório de contratos.
	Projetar serviços que possam ter uma demanda maior que a esperada.
	Serviços semelhantes devem ser consolidados em um serviço. A assinatura do serviço deve ser definida de acordo com a função mais genérica. Se as funções consideram diferentes requisitos não funcionais, então deve ser considerado o requisito funcional mais restritivo.
	Todas as funções atualizando ou modificando uma entidade (ou seja, operações de CRUD) devem ser agrupadas em um bloco de funções relacionadas à entidade.
Ausência de estado	Evite criação de serviços que guardam estados. Se necessário guardar estados, utilize a linguagem BPEL para tal.
Granularidade	Verificar se as informações processadas por um serviço estão de acordo com as informações esperadas pelos consumidores.
	Projetar serviços com granularidade baixa para disponibilizar serviços com alto valor ao negócio.

Conforme abordado por [Dirr *et al.*, 2012], o projeto dos serviços deve considerar as seguintes heurísticas:

- **Separar serviços de dados de serviço de lógica** – Devido à diferença de características, serviços de dados e de lógica devem ser separados em

diferentes projetos. Serviços de dados correspondem a operações CRUD em cada entidade do negócio e, em geral, possuem alto grau de reuso. Serviços de lógica executam operações correspondentes a regras do negócio e, normalmente, utilizam-se dos serviços de dados;

- **Projetar o modelo canônico dos serviços** – O modelo canônico contém todos os dados estruturados aderentes aos requisitos do negócio. Projetá-lo reduz necessidades de transformação de dados e negociação de formatos, além de endossar o reuso dos dados existentes;
- **Definir operações dos serviços** – Devem ser definidas as operações dos serviços que correspondem aos serviços candidatos identificados. Um serviço candidato pode corresponder a uma operação ou várias operações de um serviço bem como vários serviços candidatos podem corresponder a uma única operação de um serviço;
- **Modelagem dos serviços candidatos –** Durante esta fase, deve ser documentada toda a especificação do serviço utilizando linguagem para modelagem de sistemas, como, por exemplo, UML ou UML com perfil voltado a SOA. Isto significa que podem ser desenvolvidos: modelo de classes para entidades do negócio, interfaces dos serviços, e diagrama de atividades para execução do serviço.

Esta sequência de atividades é o que [Erl, 2008] define como um acoplamento de lógica para contrato, ou contract-first, garantindo que a lógica do negócio estará totalmente incorporada ao contrato do serviço.

2.9 Orquestração e coreografia

Composição de serviços é um conceito importante em SOA pois permite criar um novo serviço a partir da composição de serviços existentes [Josuttis,2007]. Dois conceitos básicos são empregados: Orquestração e Coreografia.

Em uma orquestra, os integrantes da banda são comandados por um maestro para produzir a música. Em SOA, orquestração de serviços possui o mesmo

princípio, onde um controlador central (maestro) coordena um conjunto de serviços, formando um serviço de mais alto nível [Josuttis, 2007].

Orquestração não é a única solução para composição de serviços, principalmente quando o processo de negócio é muito extenso, tornando árdua a tarefa de manter um controlador central coordenando inúmeras tarefas. Coreografia é uma abordagem para composição de serviços, onde nenhum serviço possui conhecimento do processo como um todo, somente quando e com qual serviço deve interagir [Juric, 2007]. A Figura 7 ilustra as diferenças entre coreografia e orquestração, onde uma orquestração funciona similar a um processo de negócio com um dos serviços atuando como controlador do processo, e uma coreografia onde os serviços chamam uns aos outros.

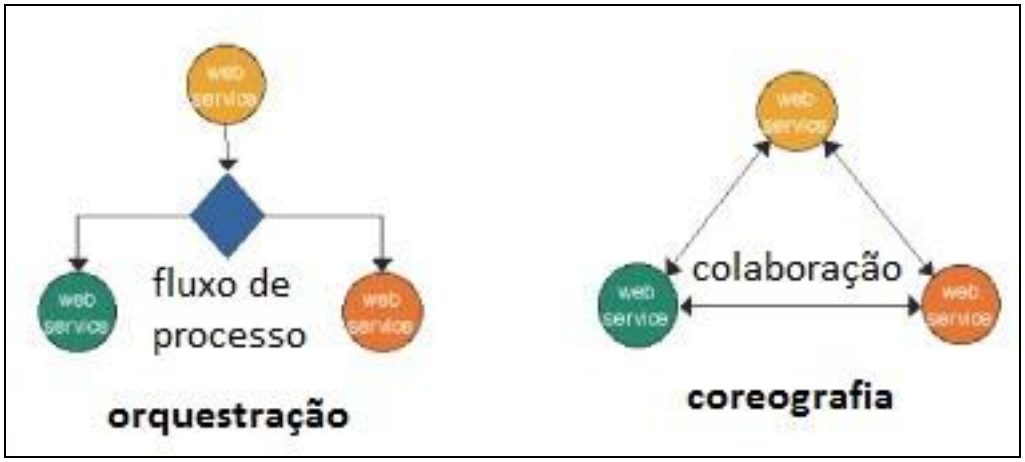

Figura 7 - Diferença entre coreografia e orquestração (adaptado de Peltz [2003])

BPEL (*Business Process Execution Language*) ou WS-BPEL é a linguagem padrão para composição de serviços implementados como web services, baseada na orquestração de serviços [Fatinato, 2007]. Ela foi definida com a junção de normas da OASIS (Organization for the Advancement of Structured Information Standards) e utiliza o padrão XML. A linguagem permite utilizar as principais vantagens de orquestração, tais como tratamento de exceção, compensação e flexibilidade de inserção de serviços [Ministro *et al.* 2011]. Apesar disso, em casos onde os serviços tenham conhecimento dos dados de saída do serviço anterior e de entrada do serviço seguinte, o uso de coreografia será o mais indicado, podendo, inclusive, coexistir ambos os tipos de composição em um ambiente distribuído.

Capítulo 3: Contract-First: projetando web services interoperáveis

Este trabalho tem o objetivo de se aprofundar no desenvolvimento de serviços com foco na abordagem *contract first*. Dessa forma, este capítulo apresenta os principais conceitos relacionados com a especificação de um bom contrato de serviço indicando as abordagens utilizadas. Detalhes técnicos são apresentados no Capítulo 4.

Dessa forma, são apresentados: conceitos de contrato de serviços; boas práticas e regras para especificação de contratos definidas pela WS-I (Web Services Interoperability Organization) na especificação do Basic Profile (BP); abordagem de centralização de esquema e como esta abordagem foi empregada neste trabalho; e padrões de projeto para criação de esquemas XML.

3.1 Contrato de serviço

Um contrato de serviço determina tudo que um serviço deve fazer de um provedor específico para um consumidor específico [Josuttis, 2007]. Desta forma, um contrato deve conter informações sobre todos os requisitos do serviço, incluindo aspectos não-funcionais, como qualidade e segurança.

Hewitt [2009] apresenta que "um contrato de serviço não é somente o WSDL, mas podem ser incluídos itens como políticas, acordos de nível de serviço (SLA), autorizações de usuário...". Assim, o conceito de contrato de serviço corresponde a uma especificação mais completa possível para que o consumidor saiba tudo que é necessário para poder utilizar o serviço.

Erl [2007] divide o contrato de serviço em duas categorias distintas: documentos técnicos e documentos adicionais, conforme Figura 8.

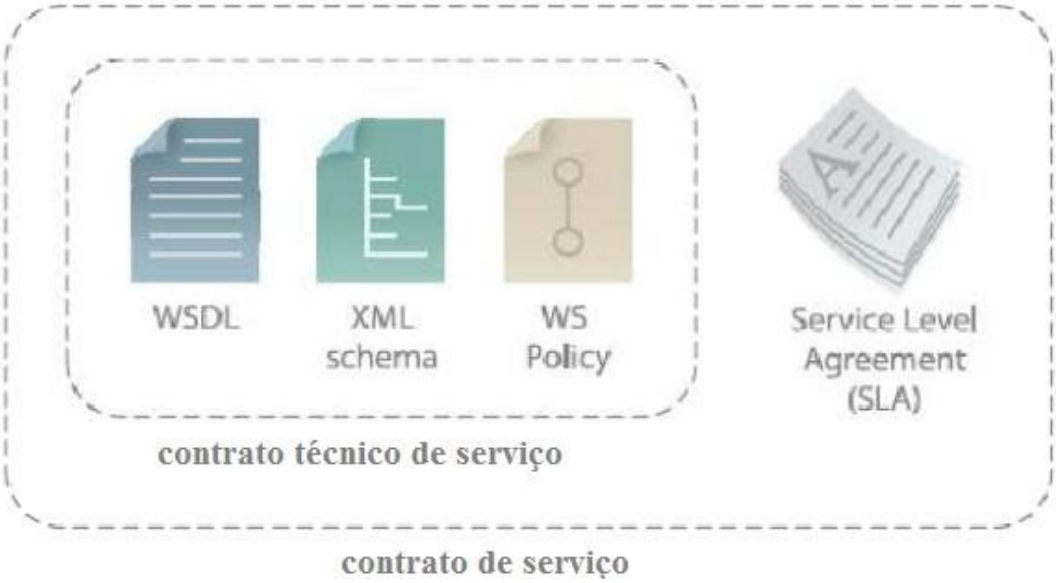

Figura 8 - Contrato de Serviço (adaptado de Erl [2007])

A documentação técnica inclui os seguintes artefatos:

- **WSDL:** descreve aspectos abstratos e concretos de serviços, provendo informações sobre suas operações, mensagens trocadas entre provedor e consumidor, o endereço físico para acessar o serviço (*endpoint*) e protocolo de transporte;
- **XSD:** contém as definições de tipos de dados necessários à troca de mensagem pelos serviços. A centralização dos tipos de dados no XSD e sua declaração como um arquivo a parte proporcionam a reutilização desses tipos por outros serviços. Para utilizar o XSD, o WSDL deve importá-lo;
- **WS-Policy:** define assertivas de políticas (regras, requisitos e obrigações) que podem ser definidas e anexadas ao WSDL. Especifica informações de qualidade de serviços e complementa a descrição do WSDL. Assim, cabe ao WSDL definir os requisitos funcionais e ao WS-Policy definir os requisitos não-funcionais.

Documentos adicionais contêm descrições não técnicas que complementam as definições técnicas do contrato. São exemplos deste tipo de documentação: contrato como qualidade de serviço (QoS) e acordo de nível de serviço (SLA).

Para serviços baseados em SOAP, o WSDL representa a parte física do contrato [Hewitt, 2009]. O método que constrói o WSDL primeiro e posteriormente codifica os requisitos em uma linguagem de programação (como, por exemplo, Java) a partir das definições presentes no WSDL é chamado *contract-first*. Em contrapartida, o método onde primeiro é feita a codificação do serviço e a partir da codificação o WSDL é gerado é chamado *code-first* ou *contract-last*.

Ambos os métodos estão relacionados à geração do WSDL e, implicitamente, à geração do XSD, que é o foco do presente trabalho. Apesar de suas importâncias no ciclo de vida de serviços, os demais componentes do contrato de serviço não são considerados neste trabalho, o qual foca o conceito de contrato especificamente à construção do WSDL.

3.2 Conformidade com WS-I Basic Profile (BP)

SOA é um paradigma para a realização e manutenção de processos de negócio em um grande ambiente de sistemas distribuídos que são controlados por diferentes proprietários [Josuttis, 2007]. A integração entre sistemas que utilizam distintas tecnologias de desenvolvimento e plataformas é viável através da utilização de serviços independentes de plataforma e protocolo. O padrão de fato para desenvolvimento de serviços é web services [Erl, 2007]. Contudo, para garantir sua utilização pelo maior número de consumidores possíveis é importante seguir boas práticas de desenvolvimento.

O desenvolvimento de web services requer utilização de uma série de especificações, como, por exemplo, XSD, WSDL, SOAP, etc. Em cada especificação, há diversas possibilidades de configuração de seus recursos. Desta forma, o desenvolvedor está livre para efetuar escolhas em uma variedade de possibilidades, tais como, camadas de transporte, políticas de segurança, mecanismos de codificação de mensagem, versão de XML utilizado, entre outros. É justamente esta combinação de possibilidades que dificulta a interoperabilidade entre serviços.

Para minimizar essas dificuldades e prover o maior nível de interoperabilidade possível, a WS-I – um consórcio composto de um grande número de fornecedores – definiu padrões para implementação de serviços. Seu produto

principal é o Basic Profile (BP), que consiste em um conjunto de padrões e boas práticas. O BP auxilia os desenvolvedores a escolherem as configurações de seus web services de forma a garantir interoperabilidade com o maior número de plataformas tecnológicas.

Hewitt [2009] sumarizou os padrões e boas práticas menos óbvios presentes no BP. Boa parte da lista indicada por Hewitt foi utilizada no presente trabalho. A seguir seguem descrições do que foi empregado:

- O elemento <body> de um <SOAP envelope> deve conter exatamente zero ou um elemento filho e deve ser *namespace qualified*, ou seja, os elementos e/ou tipos utilizados no esquema devem estar de acordo com o *namespace* especificado como *targetNamespace*. Não deve conter instruções de processamento ou utilizar DTD;
- Utilize *literal message encoding* em detrimento de SOAP *encoding*. Utilizando *literal*, os elementos e/ou tipos complexos existentes na mensagem são legíveis (literais) tanto para o consumidor quanto para o provedor, estando desacoplada de linguagem e codificação. Utilizando *encoded*, as mensagens são codificadas utilizando *SOAP-Encoding*, gerando um acoplamento com esta tecnologia;
- Apesar de ser permitido pela especificação SOAP 1.1, não se deve utilizar notação de ponto para redefinir o significado de um elemento <*faultcode*>. Essa abordagem simplifica o significado do erro. O BP indica que o detalhamento da informação contida no elemento <*faultstring*> deve ser mantido;
- A utilização de SOAP 1.1 é restrita ao protocolo HTTP. O protocolo HTTP recomendado para utilização é HTTP 1.0 ou HTTP 1.1;
- Apesar de SOAP Action ter a pretensão de auxiliar na indicação da rota de mensagem para uma determinada operação, na prática os serviços não permitem a utilização de SOAP Action, de forma que toda a informação pertinente é realizada no envelope SOAP e não em cabeçalhos HTTP. Assim, SOAP Action deve ser utilizado somente com uma dica;

- Todas as SOAP Actions devem ser especificadas no WSDL utilizando uma *string* entre aspas. Se as SOAP Actions não forem especificadas, deve-se utilizar uma *string* vazia entre aspas;
- Os cookies podem ser utilizados, mas sua utilização não é incentivada porque consumidores que não suportam cookies não estão aptos a utilizarem o seviço. Cookies não tem relação com SOAP e, por isso, não parece uma boa idéia desenvolver serviços que dependam deles. Desta forma, apesar do BP permitir sua utilização, essa tecnologia não é utilizada neste trabalho;
- Utilize XML 1.0 porque esta é a única versão que o BP oferece suporte para XML Schema e WSDL;
- Utilize codificação UTF-8 ou UTF-16;
- Tanto a especificação do WSDL quanto o BP recomendam a separação de arquivos WSDL em componentes modulares e criação de arquivos que separam a definição abstrata da definição concreta do WSDL;
- Ao importar um WSDL, deve haver coerência entre os namespaces de ambos os arquivos. Isso significa que o targetNamespace indicado no elemento <definitions> do WSDL importado deve ser o mesmo namespace utilizado no WSDL que o está importando. A Figura 9 e a Figura 10 apresentam exemplos desta boa prática do serviço TratarCliente, cujo WSDL foi dividido em dois: um para a parte abstrata do WSDL serviço e o outro para sua parte concreta. A Figura 9 apresenta à seção *definition* do WSDL abstrato que é importado pelo WSDL concreto. A Figura 9 corresponde a parte do WSDL concreto ilustrando a importação do WSDL abstrato, utilizando o mesmo namespace utilizado no WSDL abstrato;

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions name="TratarClienteDataService"
    targetNamespace="http://uniriotec.monografia.br/TratarClienteDataService/defs"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://uniriotec.monografia.br/TratarClienteDataService/defs"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:cv="http://uniriotec.monografia.br/ClienteView">
```

Figura 9 – Trecho de WSDL abstrato que é importado

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions name="TratarClienteDataService"
    targetNamespace="http://uniriotec.monografia.br/TratarClienteDataService/service"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://uniriotec.monografia.br/TratarClienteDataService/service"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:defs="http://uniriotec.monografia.br/TratarClienteDataService/defs">

    <wsdl:documentation>
        O WSDL concreto define como e onde consumir este serviço.
        Esse padrão de projeto permite fornecer diferentes possibilidades
        de binding e endpoints mantendo as definições abstratas do serviço.
    </wsdl:documentation>

    <wsdl:import namespace="http://uniriotec.monografia.br/TratarClienteDataService/defs"
            location="TratarClienteDataServiceAbstrato.wsdl" />
```

Figura 10 – Trecho de WSDL concreto que importa o WSDL abstrato

- Utilize somente os recursos proporcionados pela especificação do WSDL. A utilização de extensões de WSDL pode implicar na inviabilidade de consumo de serviços por parte de alguns consumidores que não suportam tais extensões;

- O BP não permite o uso de tipos de arrays no WSDL 1.1 devido à diversidade de interpretações de como estes devem ser utilizados. Essa diversidade implica em uma série de problemas de interoperabilidade. Uma solução alternativa simples ao uso de arrays é definir o atributo maxOccurs de um tipo complexo com um valor maior que 0. É possível determinar um número indeterminado utilizando o valor *unbounded* para maxOccurs;

- Os estilos existentes na costrução de um WSDL são *Document* e *RPC*. Quando o modelo selecionado é *Document*, as mensagens de entrada e saída representam documentos, ou seja, referem-se a um *parts* cujo tipo é *element* na seção *body* do WSDL. Quando o modelo selecionado é *RPC*, as mensagens correspondem aos parâmetros de entrada e retorno, e deve-se utilizar obrigatoriamente *types* (simples ou complexos). O BP permite a utilização de ambos os estilos;

- Web services são capazes de abtstrair o conteúdo de mensagens e operações fora da camada de transporte. Entretanto, apenas SOAP é suportado pelo BP em elementos <binding> (vide Tabela 1 para entendimento de *binding*).

O BP define boas práticas e padrões de projetos para web services. Cabe ao desenvolvedor identificar os padrões necessários a seus serviços. Essa identificação deve levar em consideração os requisitos funcionais e não-funcionais.

3.3 Centralização de Esquema

A definição dos tipos de dados a serem empregados nas mensagens trafegadas pelos serviços é uma questão essencial. A questão que surge é a centralização de esquema versus a definição de esquemas descentralizados.

A descentralização de esquema refere-se a especificação dos tipos de dados de um serviço de forma independente dos demais serviços. As vantagens neste tipo de abordagem é o fato de uma alteração em um tipo consumido ou retornado por um serviço não gerar impacto em outros serviços. No entanto, aumenta-se a necessidade de se realizar transformações de dados na comunicação entre serviços que utilizam tipos de dados com estruturas diferentes, mas que representam a mesma informação.

Erl [2008] apresenta o padrão centralização de esquema e defende a definição de um esquema "oficial" para cada conjunto de informações. Contratos de serviços podem compartilhar esses esquemas centralizados. O objetivo é padronizar os tipos de dados em um determinado domínio, seja um processo, uma indústria (financeira, farmacêutica, petrolífera, etc.) ou a organização inteira.

A centralização do esquema de dados proporciona a eliminação de tipos redundantes. Neste projeto, essa abordagem se torna viável devido ao conhecimento das informações necessárias ao processo, identificadas na modelagem do processo e pela definição das entidades e seus relacionamentos, mapeadas em um modelo canônico.

Essa abordagem aumenta a possibilidade de reutilização do serviço, garante sua interoperabilidade e aumenta sua eficiência por reduzir o uso de transformação de dados. Entretanto, aumenta o acoplamento dos serviços com os tipos de dados comuns centralizados no esquema, o que pode significar um desafio à gestão dos tipos fundamentais.

Alterações de semântica e atributos podem se tornar um sério problema para a governança de uma arquitetura orientada a serviços, conforme exemplificado abaixo:

“Vamos supor que crie um tipo de dados harmonizado para clientes. Depois, um sistema de contabilidade que pode precisar de mais dois novos atributos do cliente para lidar com diferentes taxas de juros, enquanto um sistema de CRM pode introduzir novas formas de endereços eletrônicos e um sistema de oferta pode precisar dos atributos para lidar com a proteção da privacidade. Se o tipo de dados de cliente é compartilhado por todos os seus sistemas (incluindo sistemas que não estão interessados em nenhuma dessas novidades), todos os sistemas deverão estar atualizados para refletir cada mudança, e o tipo de dado de cliente se tornará mais e mais complicado.” [Josuttis, 2007].

Neste trabalho, com o intuito de obter os benefícios relativos ao padrão centralização de esquema, mas minimizar a possibilidade de conceitos distintos em relação às entidades identificadas e mapeadas, a centralização de esquema por processo de negócio foi utilizada. Esta abordagem foi escolhida porque em um processo mapeado e bem definido, o conceito sobre uma determinada entidade é compartilhado a todos que participam deste processo.

Esta abordagem garante a interoperabilidade e eficiência dos serviços que participam do processo e diminui a possibilidade de necessidade de redefinição de uma determinada entidade apenas para um ou outro grupo de consumidores. Na prática, isso significa que todos os consumidores que usufruem de serviços de um determinado processo de análise de crédito têm o mesmo conceito sobre as entidades (por exemplo, cliente, funcionário). Se um consumidor utiliza um serviço de um processo de contabilidade para obter informações para um processo de marketing, por exemplo, é possível que os atributos relativos à entidade cliente não sejam iguais, necessitando um mapeamento prévio para garantir a transformação de dados e, assim, a interoperabilidade. Apesar de não eliminar a necessidade de transformação de dados, essa abordagem a minimiza e possibilita uma governança mais eficiente sobre os dados.

3.4 Padrões de projeto de XML Schema

Seguindo as melhores práticas de padrão de XML Schema, Hewitt [2009] propõe a utilização de alguns padrões de projeto para esquemas, buscando obter uma

mescla de coesão, acoplando, simplicidade e exposição de tipos. São eles: Boneca Russa, Fatia de Salame, Veneziana e Jardim do Éden.

- **Boneca Russa:** Consiste em um único elemento raiz global. Todos os outros tipos e elementos são aninhados ao elemento raiz, sendo possível utilizá-los somente uma única vez.
- **Fatia de Salame:** Ao contrário do padrão boneca russa, todos os elementos são globais, tornando o esquema reutilizável. Os tipos são somente locais.
- **Veneziana:** É uma extensão do padrão boneca russa. Possui apenas um elemento raiz global, entretanto os tipos são definidos externamente ao elemento, e não aninhados.
- **Jardim do Éden:** É uma combinação do padrão Fatia de Salame com o padrão Veneziana. Sua proposta é maximizar o reuso definindo todos os elementos e tipos como globais.

A Tabela 6 apresenta um comparativo entre os padrões abordados.

Tabela 6 - Comparativo entre padrões XML Schema

Padrão de Projeto	Vantagens	Desvantagens
Boneca Russa (*Russian Doll*)	• Alta Coesão e acoplamento mínimo • Facilidade de escrita e leitura do esquema	• Impossibilidade de reuso de tipos • Impossibilidade de divisão em dois ou mais arquivos
Fatia de Salame (*Salami Slice*)	• Facilidade de reuso • Suporta reuso em outros documentos	• Fortemente acoplado • Documento extenso.
Veneziana (Venetian Blind)	• Permite múltiplos arquivos • Facilidade de reuso	• Fortemente acoplado • Documento extenso
Jardim do Éden (Garden of Eden)	• Permite múltiplos arquivos • Extrema facilidade de reuso • Flexibilidade • Fácil manutenção • Baixo Acoplamento	• Difícil de ler e entender • Documento extenso • Múltiplos elementos raiz

Observando os padrões abordados e levando em conta o domínio do processo e a proposta de desenvolvimento dos serviços, será utilizado o padrão Jardim do Éden, a fim de aproveitar vantagens cruciais da proposta, como uso de múltiplos arquivos, facilidade de reuso entre vários serviços e o baixo acoplamento dos tipos

de dados. No entanto, serão realizadas adaptações para que possam ser minimizadas as desvantagens apresentadas.

Dessa forma, o modelo canônico de um determinado processo deve ser implementado considerando um esquema centralizado contendo suas entidades básicas. Além deste esquema central, deve existir um esquema distinto para cada entidade, localizado em um arquivo XSD a parte, representando seus relacionamentos e cardinalidades. A Figura 11 apresenta um exemplo de um modelo contendo as entidades Pessoa e Endereco apenas com as informações básicas de cada uma delas. Já a Figura 12 apresenta uma visão da entidade Pessoa, mostrando que há uma relação entre uma pessoa e seus endereços.

```
<xsd:complexType name="Pessoa">
    <xsd:sequence>
        <xsd:element name="cpf" type="xsd:string"/>
        <xsd:element name="nome" type="xsd:string"/>
        <xsd:element name="identidade" type="xsd:string"/>
        <xsd:element name="telefone" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Endereco">
    <xsd:sequence>
        <xsd:element name="cep" type="xsd:string"/>
        <xsd:element name="logradouro" type="xsd:int" />
        <xsd:element name="cidade" type="xsd:date" />
        <xsd:element name="estado" type="xsd:string" />
        <xsd:element name="pais" type="xsd:string" />
    </xsd:sequence>
</xsd:complexType>
```

Figura 11 - XSD com entidades básicas

```
<xsd:import namespace="http://www.example.org/ModeloCanonicoExemplo"
            schemaLocation="../ModeloCanonicoExemplo.xsd">
</xsd:import>

<xsd:complexType name="PessoaView">
    <xsd:sequence>
        <xsd:element name="pessoa" type="mc:Pessoa"
            minOccurs="1" maxOccurs="1" />
        <xsd:element name="enderecos" type="mc:Endereco"
            minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
```

Figura 12 - XSD de visão

Para realizar essa divisão entre os esquemas, devem-se considerar os serviços de candidatos identificados para avaliar que conjunto de dados deve ser usado em cada serviço. Esses esquemas foram chamados de visões. Cada esquema de visão se utiliza do esquema central para manipular os tipos de dados do domínio, maximizando o reuso dos mesmos. A abordagem permite inclusive flexibilidade de que novos esquemas de visão sejam criados, de acordo com a necessidade do serviço. Adicionalmente, as especificações das estruturas das mensagens trocadas entre serviços e seus consumidores possuem apenas informações relevantes à operação, evitando o consumo de recursos desnecessariamente.

Essa abordagem possibilita a utilização do Jardim do Éden sem o ônus relativo ao padrão, conforme demonstrado abaixo:

- **Difícil de ler e entender**: com o modelo de visões, os serviços utilizam somente os esquemas necessários. Desta forma, não é apresentado ao desenvolvedor todos os tipos complexos ou elementos relativos ao processo de análise de crédito, mas somente aqueles necessários aos esquemas que estão sendo utilizadas no WSDL;
- **Documento extenso:** como os esquemas são divididos em visões e cada visão tem somente os tipos complexos ou elementos necessários à visão, o documento fica potencialmente menor;
- **Múltiplos elementos raiz:** cada visão tem somente os tipos complexos ou elementos necessários à visão, o que diminui o número de elementos raiz passíveis de serem utilizados pelo desenvolvedor.

3.5 Modularização de WSDL

Conforme descrito na seção 2.3, o WSDL pode ser dividido em três partes distintas que correspondem à definição abstrata, a definição concreta e a documentação, que pode ser utilizada nas duas partes anteriores.

Considerando esse conceito, é possível separar a parte concreta da parte abstrata do WSDL, tornando o serviço mais modular e flexível. Essa modularização proporciona que serviços com diferentes mecanismos de transporte, codificação e *endpoint* utilizem a mesma interface e tipos de dados [Hewitt, 2009].

Além da modularização do WSDL, é possível especificar um esquema externo ao WSDL e importá-lo. Assim, a modularização do WSDL deve seguir as seguintes diretrizes:

- XSD: XSD é importado pelo arquivo que corresponde à parte abstrata do WSDL;
- WSDL abstrato: representa as interfaces do serviço, ou seja, contém o nome de cada capacidade do serviço, mensagens de requisição e resposta (*messages* e *portTypes*). A Figura 13 apresenta um exemplo de WSDL abstrato que é importado pelo WSDL abstrato;
- WSDL concreto: representa as características de implementação do serviço (elementos *binding* e *services*). A Figura 14 apresenta um exemplo de WSDL concreto que importa o WSDL concreto. É este WSDL que é disponibilizado para os consumidores quando o serviço é publicado.

```
<definitions name="StockQuote"
  targetNamespace="http://soacookbook.com/stockquote/defs"
  xmlns:tns="http://soacookbook.com/quote/defs"
  xmlns:xsd1="http://soacookbook.com/quote/schemas"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

   <import namespace="http://soacookbook.com/quote/schemas"
     location="http://soacookbook.com/quote/stockquote.xsd"/>

    <message name="GetLastTradePriceInput">
        <part name="body" element="xsd1:TradePriceRequest"/>
    </message>

    <message name="GetLastTradePriceOutput">
        <part name="body" element="xsd1:TradePrice"/>
    </message>

    <portType name="StockQuotePortType">
        <operation name="GetLastTradePrice">
           <input message="tns:GetLastTradePriceInput"/>
           <output message="tns:GetLastTradePriceOutput"/>
        </operation>
    </portType>
</definitions>
```

Figura 13 – StockQuoteAbstract.wsdl [Hewitt, 2009]

```
<definitions name="StockQuote"
  targetNamespace="http://soacookbook.com/quote/service"
  xmlns:tns="http://soacookbook.com/quote/service"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:defs="http://soacookbook.com/quote/defs"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <import namespace="http://soacookbook.com/quote/defs"
    location="http://soacookbook.com/quote/StockQuoteAbstract.wsdl"/>

    <binding name="StockQuoteSoapBinding"
        type="defs:StockQuotePortType">
        <soap:binding style="document"
          transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetLastTradePrice">
           <soap:operation
             soapAction="http://soacookbook.com/GetLastTradePrice"/>
           <input><soap:body use="literal"/></input>
           <output><soap:body use="literal"/></output>
        </operation>
    </binding>

    <service name="StockQuoteService">
        <documentation>Returns the current price for a given ticker.
        </documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteBinding">
           <soap:address location="http://soacookbook.com/quote"/>
        </port>
    </service>
</definitions>
```

Figura 14 - StockQuoteConcrete.wsdl [Hewitt, 2009]

3.6 Contract-first versus code-first

Na abordagem code-first, primeiramente é desenvolvida a codificação e, em seguida, gera-se o contrato do serviço com auxílio de um framework. Essa abordagem é mais utilizada devido a sua praticidade [Kalin, 2009]. O desenvolvedor não precisa conhecer e desenvolver a documentação técnica do contrato de serviços (WSDL, XSD e WS-Policy) e, por isso, o desenvolvimento é mais rápido. Em projetos pequenos, onde previamente não foram identificadas possibilidades de reutilização em escala do serviço, essa abordagem é suficiente. Contudo, em projetos de médio a grande porte, onde se planeja utilizar SOA ou a mesma já está em execução, essa abordagem pode interferir em princípios básicos da arquitetura como: alinhamento com o negócio, interoperabilidade, acoplamento, reuso e durabilidade.

A abordagem contract-first é menos difundida. Possíveis explicações para esse fato são: maior complexidade de desenvolvimento devido à necessidade de domínio sobre elementos diferentes da linguagem de programação usualmente utilizada e pouca difusão das técnicas e benefícios da abordagem contract-first. Em

contract-first, primeiramente elabora-se o contrato do serviço e, em seguida, utiliza-se desse contrato para gerar a codificação correspondente. Como a codificação depende do contrato, observa-se um acoplamento positivo da lógica (codificação) para o contrato (WSDL, XSD e WS-Policy).

3.6.1 Acoplamento

Ao desenvolver primeiramente o contrato, obrigatoriamente a codificação deve considerar o que foi especificado no WSDL, XSD e WS-Policy. Do ponto de vista da arquitetura orientada a serviços, este acoplamento de lógica para contrato é positivo [Erl, 2008] e garante o mesmo contrato independentemente da linguagem de desenvolvimento ou mesmo framework utilizado. É preciso ter em mente que o foco de desenvolvimento de SOA é o XML que possibilita interoperabilidade entre sistemas distribuídos independentemente de plataforma e não na codificação de uma linguagem de desenvolvimento específica como, por exemplo, Java.

Quando se codifica primeiro, não é possível determinar completamente como o contrato será gerado. Para serviços já disponibilizados, essa abordagem traz uma dependência significativa à tecnologia utilizada para manutenção do serviço. Na prática, significa que se for identificada a necessidade de trocar a tecnologia de desenvolvimento de serviços, será necessária também a alteração do contrato, o que afeta diretamente todos os consumidores destes serviços.

Em empresas de médio e grande porte, essa substituição pode ocorrer com maior frequência por fatores diversos como: substituição de tecnologia proprietária por tecnologia open source ou vice-versa, substituição de tecnologia em função de desempenho ou facilidade de programação, entre outros. Todas as empresas que utilizam SOA devem estar preparadas para alterar a tecnologia de desenvolvimento sem afetar seus consumidores finais e o uso da abordagem *contract-first* é a ideal para esse objetivo.

Com o intuito de exemplificar o que foi discutido até o momento, foram desenvolvidos dois serviços em Java que utilizam o mesmo código (um simples serviço *HelloWorld*), mas com frameworks diferentes (Axis e JAX-WS) que foram implantados (*deploy*) no mesmo container web. Mesmo com um exemplo tão

simplificado, é possível identificar as diferenças nos contratos gerados com a utilização frameworks diferentes. A seguir, é apresentado o detalhamento das diferenças apresentadas nos contratos gerados automaticamente por ambos os frameworks, divididas nas seções do WSDL.

Definições:

A Figura 15 e a Figura 16 dividem as definições em dois grupos distintos. O grupo 1 corresponde às definições geradas pelos dois frameworks, ou seja, contém os mesmos *namespaces*. O grupo 2 corresponde às definições geradas por apenas um dos frameworks, ou seja, possui *namespaces* que foram gerados em um contrato, mas não estão presentes no outro.

Focando a análise no grupo 1, é possível identificar diferenças sensíveis na nomenclatura dos prefixos de cada definição, o que implica na impossibilidade de reutilização do contrato de um se a implementação for substituída pelo outro.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <wsdl:definitions
3      targetNamespace="http://uniriotec.monografia.br"
4      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
5      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
6      xmlns:impl="http://uniriotec.monografia.br"
7      xmlns:intf="http://uniriotec.monografia.br"
8      xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
9
10     xmlns:apachesoap="http://xml.apache.org/xml-soap"
11      >
```

(1) (2)

Figura 15 - Definitions geradas pelo Axis 1.4

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <definitions
3      targetNamespace="http://uniriotec.monografia.br/"
4      xmlns="http://schemas.xmlsoap.org/wsdl/"
5      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
6      xmlns:tns="http://uniriotec.monografia.br/"
7      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
8
9      xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
10     xmlns:wsp="http://www.w3.org/ns/ws-policy"
11     xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
12     xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
13     name="HelloWorldImplService"
14      >
```

(1) (2)

Figura 16 - Definitions geradas pelo JAX-WS 2.2

Tipos:

A mesma classe foi utilizada para desenvolver os serviços nos dois frameworks. Esta consiste em apenas uma função que recebe como parâmetro um atributo chamado *nome*, cujo tipo é uma *String*, e retorna também uma *String*, conforme exibido na Figura 17.

```
1  package br.monografia.uniriotec;
2
3  import javax.jws.WebService;
4
5  @WebService
6  public interface IHelloWorld {
7      public String getHelloWorld(String nome);
8  }
9
```

Figura 17 - Interface do serviço HelloWorld utilizado pelo AXIS e JAX-WS

Os tipos de entrada e saída foram gerados em ambos os contratos a partir dos atributos da classe Java. A Figura 18 apresenta a geração de tipos empregando o framework Axis. Foram gerados os elementos *getHelloWorld* para o parâmetro *nome* de entrada do método e *getHelloWorldResponse* para o retorno cujo tipo é *String*. Em ambos os casos, a estrutura do tipo complexo foi definida dentro do próprio elemento. Isto não permite o reuso de tipos. Já a Figura 19 apresenta os tipos gerados empregando o JAX-WS. Foram gerados também os elementos *getHelloWorld* e *getHelloWorldResponse*. No entanto, diferentemente do Axis, a definição dos tipos foi feita fora dos elementos.

Além disso, os atributos que representam os tipos complexos utilizados pelos elementos na requisição e resposta ficaram com nomenclaturas diferentes. No Axis, o atributo do tipo complexo utilizado na requisição foi nomeado para *nome* e o atributo do tipo complexo utilizado como retorno foi nomeado para *getHelloWorldReturn*, No JAX-WS, diferentemente do Axis, o atributo do tipo complexo utilizado na requisição foi nomeado para *arg0* e o atributo do tipo complexo utilizado como retorno foi nomeado para *return*. Dessa forma, o cliente do serviço seria fortemente

impactado apenas pela substituição do framework de disponibilização do serviço, mesmo sem alterar o código do mesmo, pois teria que considerar um atributo *nome* ao invés de *arg0*, por exemplo.

Vale ressaltar ainda que a forma de definição dos tipos de dados no WSDL foi distinta nos dois frameworks. O Axis gera os tipos de dados inline (ou seja, diretamente no arquivo WSDL) enquanto que o JAX-WS gera um arquivo XML Schema a parte e o importa no WSDL gerado.

```
<wsdl:types>
    <schema elementFormDefault="qualified"
        targetNamespace="http://uniriotec.monografia.br"
        xmlns="http://www.w3.org/2001/XMLSchema">
        <element name="getHelloWorld">
            <complexType>
                <sequence>
                    <element name="nome" type="xsd:string" />
                </sequence>
            </complexType>
        </element>
        <element name="getHelloWorldResponse">
            <complexType>
                <sequence>
                    <element name="getHelloWorldReturn" type="xsd:string" />
                </sequence>
            </complexType>
        </element>
    </schema>
</wsdl:types>
```

Figura 18 - Tipos definidos no WSDL pelo Axis 1.4

```
<?xml version="1.0" encoding="UTF-8" ?>

<xs:schema xmlns:tns="http://uniriotec.monografia.br/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0"
    targetNamespace="http://uniriotec.monografia.br/">
    <xs:element name="getHelloWorld" type="tns:getHelloWorld" />
    <xs:element name="getHelloWorldResponse" type="tns:getHelloWorldResponse" />
    <xs:complexType name="getHelloWorld">
        <xs:sequence>
            <xs:element name="arg0" type="xs:string" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="getHelloWorldResponse">
        <xs:sequence>
            <xs:element name="return" type="xs:string" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Figura 19 - Tipos definidos em XSD e importados pelo WSDL no JAX-WS 2.2

Mensagens:

No caso das mensagens, a diferença apareceu apenas na nomenclatura da mensagem de requisição. No Axis, a mensagem de requisição foi nomeada para *getHelloWorldRequest* (Figura 20), enquanto que no JAX-WS foi nomeada para *getHelloWorld* (Figura 21). Esta diferença impossibilita a continuidade do consumo do serviço.

```
<wsdl:message name="getHelloWorldResponse">
    <wsdl:part element="impl:getHelloWorldResponse"
    name="parameters" />
</wsdl:message>

<wsdl:message name="getHelloWorldRequest">
    <wsdl:part element="impl:getHelloWorld"
    name="parameters" />
</wsdl:message>
```

Figura 20 - Mensagens e PortType no Axis 1.4

```
<message name="getHelloWorld">
    <part name="parameters" element="tns:getHelloWorld" />
</message>

<message name="getHelloWorldResponse">
    <part name="parameters" element="tns:getHelloWorldResponse" />
</message>
```

Figura 21 - Mensagem e PortType no JAX-WS 2.2

Operações:

No caso das operações, a diferença apareceu também na nomenclatura da mensagem de requisição. No Axis, o nome da mensagem é *getHelloWorldRequest* (Figura 22), enquanto que no JAX-WS é *getHelloWorld* (Figura 23). Esta diferença também impossibilita a continuidade do consumo do serviço.

```
<wsdl:portType name="HelloWorldImpl">
    <wsdl:operation name="getHelloWorld">
        <wsdl:input
            message="impl:getHelloWorldRequest"
            name="getHelloWorldRequest" />
        <wsdl:output
            message="impl:getHelloWorldResponse"
            name="getHelloWorldResponse" />
    </wsdl:operation>
</wsdl:portType>
```

Figura 22 - Mensagens e PortType no Axis 1.4

```
<portType name="HelloWorldImpl">
    <operation name="getHelloWorld">
        <input
            wsam:Action="http://uniriotec.monografia.br/HelloWorldImpl/getHelloWorldRequest"
            message="tns:getHelloWorld" />
        <output
            wsam:Action="http://uniriotec.monografia.br/HelloWorldImpl/getHelloWorldResponse"
            message="tns:getHelloWorldResponse" />
    </operation>
</portType>
```

Figura 23 - Mensagem e PortType no JAX-WS 2.2

3.6.2 Desempenho

Quando um consumidor solicitar a execução de uma operação, os parâmetros utilizados devem estar de acordo com o estipulado no XSD. Caso contrário, o consumidor receberá uma mensagem de erro informando o ocorrido. O XSD valida as restrições dos tipos de dados do serviço como, por exemplo:

- Relacionamento entre os tipos de dados;
- Cardinalidade dos relacionamentos;
- Obrigatoriedade de tipos de dados;
- Lista de valores válidos para um determinado tipo de dado;
- Tamanhos e formatos específicos para tipos de dados.

Na abordagem code-first, o XSD é gerado a partir dos tipos de dados implementados para o contrato do serviço em uma linguagem de programação como, por exemplo, Java. Algumas restrições de tipos de dados podem ser expressas nas estruturas de dados que representam as entidades e algumas dessas restrições exigem

desenvolvimento para serem verificadas. O relacionamento entre as entidades e as cardinalidades *0 x 1* (relacionamento com outra entidade) e *0 x N* (lista de outra entidade) podem ser expressas diretamente nas classes que representam os tipos de dados. Contudo, cardinalidades mais precisas (por exemplo, cardinalidade de 1 a 3) devem ser validadas através de codificação uma vez que não são especificadas no WSDL gerado. Da mesma forma, validações de obrigatoriedade, valores válidos, tamanhos e formatos devem ter suas validações implementadas.

Portanto, em code-first, essas estruturas de dados que representam as entidades dão origem aos tipos de dados do WSDL, seja diretamente no WSDL (inline) ou em um XSD (sendo importado). Como grande parte das restrições não consta diretamente nas estruturas, mas em funções programadas, os tipos de dados gerados automaticamente não contêm essas restrições. Isso implica que nessa abordagem o consumidor solicita a execução de uma operação que, após verificação das restrições básicas no XSD, é encaminhada ao servidor de aplicação. O servidor de aplicação inicia uma instância do serviço e verifica se as restrições programadas foram atendidas. Se uma dessas restrições não for atendida, o serviço retorna uma mensagem de erro que é recebida pelo consumidor. Ou seja, parte da validação ocorre no XSD e outra parte em uma instância do serviço.

O desempenho de SOA torna-se crítico normalmente por causa do tempo de execução dos serviços [Josuttis, 2007]. Assim, qualquer processamento que possa ser evitado deve ser considerado. Neste caso, na teoria, a utilização de *contract-first* mostra-se como uma melhor abordagem, pois permite incluir outras restrições na estrutura das mensagens de entrada e saída das operações dos serviços. Dessa forma, um serviço só é instanciado se a mensagem sendo enviada passa pela validação segundo as regras do esquema.

3.6.3 Alteração do contrato

Alteração na interface da codificação (nomenclatura de funções e parâmetros) ou alteração na nomenclatura de atributos de classes pode ser necessária para padronização, para melhorar a semântica dos mesmos ou por qualquer outro motivo.

Se a alteração no desenvolvimento se fizer necessária em serviços já disponibilizados, idealmente, os consumidores não devem ser afetados.

Utilizando code-first, qualquer alteração na interface da codificação implicará em alteração do contrato [Kalin, 2009], fato que é contornável utilizando contract-first quando esta não envolve mudanças de tipos de dados de entrada e saída das interfaces. Havendo a necessidade de alteração dos tipos de dados, é importante que a organização tenha uma política de versionamento para acolher a nova demanda para novos consumidores sem afetar os antigos consumidores do serviço [Josuttis, 2007].

A seguir, será demonstrada como alteração na interface da codificação sem alteração dos tipos de dados de entrada e saída é tratada diferentemente em contract-first e code-first.

A Figura 24 exibe os tipos de dados e a Figura 25 exibe a interface do serviço HelloWorld desenvolvido com o framework JAX-WS 2.2 e utilizado na seção 3.6.1. Para explicitar a diferença entre as abordagens code-first e contract-first, considere que a manutenção desse serviço levou às seguintes alterações:

- Criação da classe chamada *HelloBean* responsável pela construção da mensagem de saudação;
- Alteração do nome da operação *getHelloWorld(String nome)* para *sayHello(String name)* na interface do serviço;
- Criação da interface *IHelloWorld*;
- Alterações necessárias na implementação do serviço, dadas as mudanças acima.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is
    JAX-WS RI 2.2.6b21 svn-revision#12959. -->
<xs:schema xmlns:tns="http://uniriotec.monografia.br/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0"
    targetNamespace="http://uniriotec.monografia.br/">
    <xs:element name="getHelloWorld" type="tns:getHelloWorld" />
    <xs:element name="getHelloWorldResponse" type="tns:getHelloWorldResponse" />
    <xs:complexType name="getHelloWorld">
        <xs:sequence>
            <xs:element name="arg0" type="xs:string" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="getHelloWorldResponse">
        <xs:sequence>
            <xs:element name="return" type="xs:string" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Figura 24 - XSD do serviço HelloWorld antes da manutenção do serviço

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!-- Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is
    JAX-WS RI 2.2.6b21 svn-revision#12959. -->
<!-- Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is
    JAX-WS RI 2.2.6b21 svn-revision#12959. -->
<definitions
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
    xmlns:wsp="http://www.w3.org/ns/ws-policy" xmlns:wsp1_2="http://schemas.xmlsoap.org/ws/2004/09/policy"
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://uniriotec.monografia.br/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace="http://uniriotec.monografia.br/"
    name="HelloWorldImplService">
    <types>
        <xsd:schema>
            <xsd:import namespace="http://uniriotec.monografia.br/"
                schemaLocation="http://localhost:8080/HelloWorldJAX/hello?xsd=1" />
        </xsd:schema>
    </types>
    <message name="getHelloWorld">
        <part name="parameters" element="tns:getHelloWorld" />
    </message>
    <message name="getHelloWorldResponse">
        <part name="parameters" element="tns:getHelloWorldResponse" />
    </message>
    <portType name="HelloWorldImpl">
        <operation name="getHelloWorld">
            <input
                wsam:Action="http://uniriotec.monografia.br/HelloWorldImpl/getHelloWorldRequest"
                message="tns:getHelloWorld" />
            <output
                wsam:Action="http://uniriotec.monografia.br/HelloWorldImpl/getHelloWorldResponse"
                message="tns:getHelloWorldResponse" />
        </operation>
    </portType>
    <binding name="HelloWorldImplPortBinding" type="tns:HelloWorldImpl">
        <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
            style="document" />
        <operation name="getHelloWorld">
            <soap:operation soapAction="" />
            <input>
                <soap:body use="literal" />
            </input>
            <output>
                <soap:body use="literal" />
            </output>
        </operation>
    </binding>
    <service name="HelloWorldImplService">
        <port name="HelloWorldImplPort" binding="tns:HelloWorldImplPortBinding">
            <soap:address location="http://localhost:8080/HelloWorldJAX/hello" />
        </port>
    </service>
</definitions>
```

Figura 25 - WSDL do serviço HelloWorld antes da manutenção do serviço

A Figura 26 apresenta as alterações no serviço considerando a abordagem *code-first*. Conforme esperado, o XSD e o WSDL gerados a partir deste código diferem dos arquivos correspondentes à implementação original. A Figura 27 ressalta nos retângulos 1 e 2 as diferenças entre os tipos de dados definidos no XSD original e novo XSD, enquanto que a Figura 28 ressalta nos retângulos 1, 2 e 3 as diferenças no WSDL.

```java
package br.monografia.uniriotec;

public class HelloBean {
    private String greeting = "Hello, ";

    public String getGreeting() {
        return greeting;
    }

    public void setGreeting(String greeting) {
        this.greeting = greeting;
    }
}
```

```java
package br.monografia.uniriotec;

import javax.jws.WebService;

@WebService
public interface IHelloWorld {
    public String sayHello(String name);
}
```

```java
package br.monografia.uniriotec;

import javax.jws.WebService;

@WebService
public class HelloWorldImpl implements IHelloWorld{

    @Override
    public String sayHello(String name) {
        return new HelloBean().getGreeting() + name;
    }
}
```

Figura 26 - Nova especificação do serviço HelloWorld - Code-First

```xml
<?xml version="1.0" encoding="UTF-8" ?>

<xs:schema xmlns:tns="http://uniriotec.monografia.br/"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0"
    targetNamespace="http://uniriotec.monografia.br/">                    1
    <xs:element name="getHelloWorld" type="tns:getHelloWorld" />
    <xs:element name="getHelloWorldResponse" type="tns:getHelloWorldResponse" />
    <xs:complexType name="getHelloWorld">
        <xs:sequence>
            <xs:element name="arg0" type="xs:string" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="getHelloWorldResponse">    2
        <xs:sequence>
            <xs:element name="return" type="xs:string" minOccurs="0" />
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

XSD original

```xml
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!-- Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is
3      JAX-WS RI 2.2.6b21 svn-revision#12959. -->
4  <xs:schema xmlns:tns="http://uniriotec.monografia.br/"
5      xmlns:xs="http://www.w3.org/2001/XMLSchema" version="1.0"
6      targetNamespace="http://uniriotec.monografia.br/">                  1
7      <xs:element name="sayHello" type="tns:sayHello" />
8      <xs:element name="sayHelloResponse" type="tns:sayHelloResponse" />
9      <xs:complexType name="sayHello">
10         <xs:sequence>
11             <xs:element name="arg0" type="xs:string" minOccurs="0" />
12         </xs:sequence>
13     </xs:complexType>
14     <xs:complexType name="sayHelloResponse">    2
15         <xs:sequence>
16             <xs:element name="return" type="xs:string" minOccurs="0" />
17         </xs:sequence>
18     </xs:complexType>
19 </xs:schema>
```

XSD gerado após manutenção do serviço

Figura 27 - XSD original versus XSD após manutenção com code-first

Figura 28 - WSDL original versus WSDL gerado após manutenção com code-first

Utilizando contract-first, a primeira coisa a ser feita é desenvolver o XSD e o WSDL. Entretanto, neste exemplo, o serviço já está em produção e, portanto, tanto o XSD quanto o WSDL já existem. Logo, as alterações necessárias do serviço serão realizadas sem alterar o contrato (elementos e tipos de dados, nome de operações, mensagens, etc.) e, consequentemente, sem afetar os consumidores deste serviço.

Desta forma, o XSD e o WSDL do serviço publicado apresentados na Figura 24 e Figura 25 não serão alterados após a manutenção do código do serviço. Isto é feito através do uso de anotações no nome das operações e dos tipos implementados no serviço para mapear os nomes das operações e dos os tipos de dados definidos no XSD e WSDL. A Figura 29, apresenta este mapeamento. Por exemplo, a operação

implementada no serviço *sayHello* é mapeada na operação *getHelloWorld* do WSDL; o parâmetro de entrada *name* é mapeado no parâmetro *parameters* da mensagem de requisição do WSDL; e o retorno da função é mapeado no parâmetro *parameters* da mensagem de resposta do WSDL. Esse mapeamento permite que o contrato não seja modificado, ou seja, que o XSD e o WSDL não sejam alterados em função da manutenção do serviço. Essa abordagem possibilita que os consumidores continuem utilizando o serviço com o mesmo WSDL e mesmo XSD, garantindo que os consumidores não serão afetados.

```java
package br.monografia.uniriotec;

public class HelloBean {
    private String greeting = "Hello, ";

    public String getGreeting() {
        return greeting;
    }

    public void setGreeting(String greeting) {
        this.greeting = greeting;
    }
}
```

```java
package br.monografia.uniriotec;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;

@WebService
public interface IHelloWorld {

    @WebMethod(operationName="getHelloWorld")
    @WebResult(name="parameters")
    public String sayHello(@WebParam(name="parameters") String name);
}
```

```java
package br.monografia.uniriotec;

import javax.jws.WebService;

@WebService(
        endpointInterface = "br.monografia.uniriotec.IHelloWorld",
        name = "HelloWorldImplService",
        targetNamespace = "http://uniriotec.monografia.br",
        serviceName = "HelloWorldImplService",
        portName = "HelloWorldImplPort",
        wsdlLocation = "WEB-INF/wsdl/HelloWorld.wsdl"
)
public class HelloWorldImpl implements IHelloWorld{

    @Override
    public String sayHello(String name) {
        return new HelloBean().getGreeting() + name;
    }
}
```

Figura 29- Nova especificação do serviço HelloWorld - Contract-First

3.6.4 Gestão de tipos de dados e WSDL

Em code-first, tanto os tipos de dados quanto o WSDL são gerados automaticamente a partir da codificação. Os tipos de dados são gerados diretamente no WSDL (in line) ou em um arquivo XSD que é importado para o WSDL. Essa

abordagem impossibilita a modularização do WSDL e a reutilização de tipos de dados gerados por um serviço em outro serviço.

Conforme descrito na seção 3.3, a centralização de esquemas de tipos de dados é importante devido aos seguintes fatores:

- Proporciona reutilização de tipos de dados;
- Melhora a interoperabilidade entre serviços por destacar transformações de dados de tipos comuns;
- Possibilita a gestão dos tipos de dados da organização e, consequentemente, o versionamento dos mesmos.

Conforme apresentado na seção 3.5, a modularização do WSDL é importante devido aos seguintes fatores:

- Proporciona o desenvolvimento de serviços com diferentes mecanismos de transporte, codificação e *endpoint* utilizando uma definição única da interface e mesmos tipos de dados;
- Ajuda na interoperabilidade, pois permite oferecer aos consumidores diferentes configurações concretas (protocolo SOAP, codificação, etc.) para a mesma interface.

Uma vez que não é possível alterar o contrato do serviço quando este é feito automaticamente, somente utilizando a abordagem contract-first é possível reutilizar tipos de dados de esquemas já existentes e obter os benefícios de contratos de serviços modularizados.

Capítulo 4: Provendo serviços interoperáveis

Esta seção apresenta o desenvolvimento dos serviços de análise de crédito seguindo o modelo de ciclo de vida orientado à *stakeholders*de Gu e Lago [2007] nas fases de projeto e implementação dos serviços, tendo como base o trabalho de Souza *et al.*[2011]. Souza *et al.* realizaram a aplicação dos métodos propostos por Azevedo *et al.* [2009a, 2009b, 2011] para identificação e análise de serviços candidatos em uma abordagem SOA resultando na identificação de 55 serviços candidatos a partir do processo de Análise de Crédito.

O modelo de processos foi elaborado em linguagem EPC (Evend-Driven Process Chain) [Scheer, 2000] é apresentado na Figura 30. O modelo completo incluindo FAD (Function Allocation Diagram), descrição de regras e de requisitos de negócio dentre outras descrições é apresentado por Diirr *et al.* [2010]. Este processo é responsável por analisar propostas de crédito, as quais podem ser aprovadas ou rejeitadas. Quando uma proposta de crédito é recebida, o cadastro do cliente é checado e o sistema verifica se o limite de crédito do cliente é suficiente para a concessão do crédito proposto. Se o limite for aprovado, então o sistema calcula as taxas do contrato para gerar uma proposta de contrato. Esta proposta de contrato é encaminhada a um analista de crédito que identifica necessidades de ajustes e o nível do risco inerente ao empréstimo. Se o contrato for aceitável, o cliente é contatado para avaliar o contrato. Uma vez que o contrato é aprovado, ele será ratificado. Se o contrato não for aprovado, ele será cancelado.

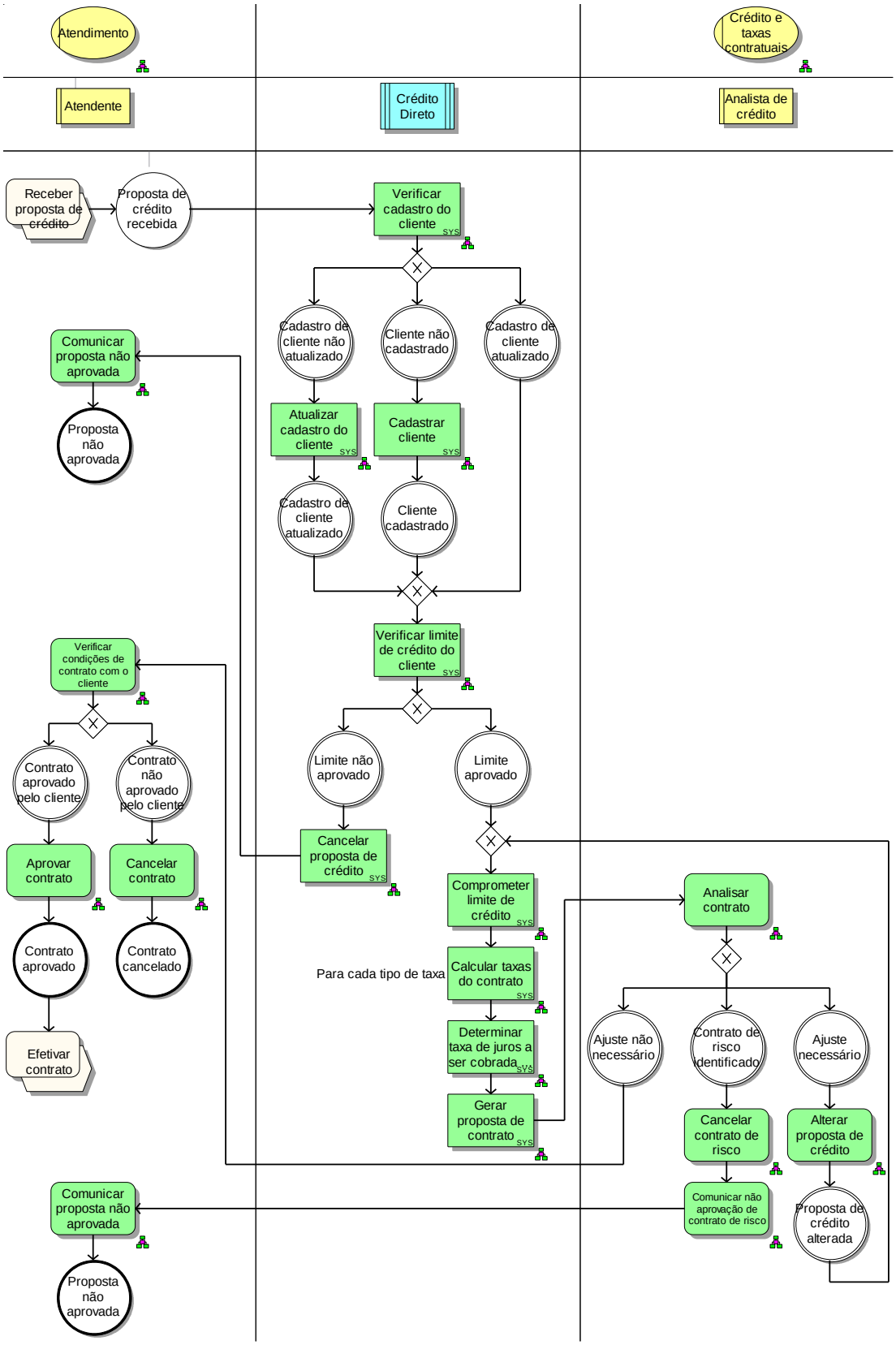

Figura 30 – Modelo em notação EPC do processo “Analisar pedido de crédito”

Os serviços identificados por Souza *et al.* [2011] a partir da aplicação das heurísticas de identificação de serviços sobre os elementos do processo (incluindo EPC, FAD, atividades, regras de negócio, requisitos de negócio etc.) foram classificados e catalogados nos serviços apresentados na Tabela 7.

Tabela 7 - Serviços candidatos identificados por Souza *et al*. [2011]

Código	Serviço Candidato	Tipo de Heurística
1	Aprovar crédito	Heurística de regras de negócio
2	Verificar limite de crédito	
3	Valor limite máximo para taxa de juros	
4	Determinar taxa de juros	
5	Verificar cliente novo	
6	Verificar cadastro de cliente desatualizado	
7	Comprometer limite de crédito_RN	
8	Ajustar proposta de crédito	
9	Identificar contrato de risco	
10	Alterar proposta de crédito	
11	Calcular taxa de impressão	
12	Calcular taxa de entrega	
13	Calcular IOF	
14	Calcular taxa de alteração do contrato	
15	Atualizar informações do cliente	Heurística de requisitos de negócio
16	Incluir cliente	
17	Verificar limite de crédito do cliente	
18	Cancelar proposta de crédito	
19	Registrar taxa de juros	
20	Gerar proposta de contrato	
21	Consultar proposta de crédito	
22	Consultar informações do cliente	
23	Comprometer limite de crédito_RqN	
24	Consultar proposta de contrato	
25	Registrar análise de contrato	
26	Consultar limite de crédito do cliente	
27	Consultar proposta de contrato para análise	
28	Consultar créditos concedidos	
29	Consultar proposta de crédito não aprovada	
30	Aprovar créditos concedidos	
31	Cancelar créditos concedidos	
32	Consultar créditos de contrato de risco	
33	Cancelar contrato de risco	
34	Calcular taxas do contrato_RqN	
35	Registrar taxas do contrato	
36	Gerar contrato com ajuste	Heurística de loop
37	Calcular taxas do contrato	Heurística de atividades de múltiplas instâncias
38	Enviar mensagem para Calcular taxas do contrato	
39	Consolidar mensagens de Calcular taxas do contrato	
40	Analisar pedido de crédito (Recebe) Receber proposta de crédito	Heurística de interfaces de processo
41	Gerar contrato	Heurística de atividades sequenciais
42	Rejeitar proposta de crédito	
43	Rejeitar contrato de risco	
44	Tratar créditos concedidos	Heurística de *cluster*

45	Tratar proposta de contrato	
46	Tratar proposta de crédito	
47	Tratar limite de crédito do cliente	
48	Tratar cadastro do cliente	
49	Tratar taxas do contrato	
50	Tratar taxa de juros do cliente	
51	Tratar taxa de juros	
52	Manter cadastro do cliente	Heurística de XOR
53	Analisar pedido de crédito	
54	Verificar contrato sem ajuste	
55	Analisar Pedido de Crédito com Proposta Aprovada	

Em seguida, Souza *et al*. [2011] aplicaram as heurísticas de análise de serviços candidatos obtendo: a priorização dos serviços, os mapas de granularidade, o modelo canônico e agrupamento de serviços de dados. A Tabela 8 apresenta os serviços candidatos para os quais a aplicação do método indicou que não deveriam ser desenvolvidos. Pequenos ajustes foram realizados nesta tabela para incluir serviços que consideramos que também não deveriam ser implementados. Já a Tabela 9 apresenta os serviços candidatos que a aplicação do método indicou para serem desenvolvidos.

Tabela 8 - Serviços candidatos indicados para não serem desenvolvidos [Souza *et al*., 2011]

Código	Serviço Candidato	Motivo da Exclusão
07	Comprometer limite de crédito_RN	Estas operações já são realizadas pelo serviço "44 - Tratar créditos concedidos".
23	Comprometer limite de crédito_RqN	
15	Atualizar informações do cliente	Foi identificado a partir da heurística de requisitos de negócio, mas corresponde à operação CRUD do serviço de dados.
16	Incluir cliente	Foi identificado a partir da heurística de requisitos de negócio, mas corresponde à operação CRUD do serviço de dados.
17	Verificar limite de crédito do cliente	Foi identificado a partir de um requisito de negócio que descreve que a regra referente ao serviço "01 – Aprovar Crédito" deve ser implementada.
22	Consultar informações do cliente	Foi identificado a partir da heurística de requisitos de negócio, mas corresponde à operação CRUD do serviço de dados.
47	Tratar limite de crédito do cliente	O limite de crédito não será armazenado, mas sim calculado.
48	Tratar cadastro do cliente	Foi identificado a partir da heurística de requisitos de negócio, mas corresponde à operação CRUD do serviço de dados.
26	Consultar limite de crédito do cliente	A consulta do limite de crédito do cliente é realizada por "01 – Aprovar crédito" e "02 - Verificar limite de crédito".
18	Cancelar proposta de crédito	Referem-se à atualização do "resultado da verificação" na entidade proposta de crédito e à consulta da proposta de crédito. Estas operações já
21	Consultar proposta de crédito	
29	Consultar proposta de crédito não	

	aprovada	são realizadas pelo serviço “45 - Tratar proposta de crédito”.
35	Registrar taxas do contrato	Esta atividade de inclusão já é feita pelo serviço “49 - Tratar taxas de contrato”.
50	Tratar taxa de juros do cliente	O serviço “44 – Tratar créditos concedidos” é responsável pela atualização da taxa de juros do cliente.
19	Registrar taxa de juro	Esta atividade de inclusão já é feita pelo serviço “44 - Tratar créditos concedidos”.
24	Consultar proposta de contrato	O registro da análise e a consulta são realizados pelo serviço “45 – Tratar proposta de contrato”.
25	Registrar análise de contrato	
27	Consultar proposta de contrato para análise	
28	Consultar créditos concedidos	A consulta e a atualização da situação dos Créditos concedidos são realizadas pelo serviço “44 – Tratar créditos concedidos”.
30	Aprovar créditos concedidos	
31	Cancelar créditos concedidos	
33	Cancelar contrato de risco	
32	Consultar créditos de contrato de risco	A consulta é realizada pelo serviço “46 – Tratar proposta de crédito”.
40	Analisar pedido de crédito (Recebe) Receber proposta de crédito	Não foi modelado o processo “Receber proposta de crédito” que utilizaria essa mensagem para invocar o processo “Analisar pedido de crédito”.
41	Gerar contrato	Já está sendo previsto no serviço “36 – Gerar contrato com ajuste”.
42	Rejeitar proposta de crédito	São fluxos muito simples para serem orquestrados.
43	Rejeitar contrato de risco	
55	Analisar pedido de crédito com proposta aprovada	Deixa uma parte do processo de fora, que o serviço “53 – Analisar pedido de crédito” contempla.

Tabela 9 - Serviços candidatos indicados para serem desenvolvidos [Souza *et al.*, 2011]

Agrupamento	Índice	Serviços	Tipo
Verificar limite de crédito	1	Aprovar crédito	Lógica
	2	Verificar limite de crédito	Lógica
Tratar taxa de juros	3	Valor limite máximo para taxa de juros	Lógica
	4	Determinar taxa de juros	Lógica
	51	Tratar taxas de juros	Dados
Manter cadastro do cliente	5	Verificar cliente novo	Lógica
	6	Verificar cadastro de cliente desatualizado	Lógica
	48	Tratar cadastro do cliente	Dados
	52	Manter cadastro do cliente	Lógica/Dados
Verificar contrato de risco	8	Ajustar proposta de crédito	Lógica
	9	Identificar contrato de risco	Lógica
	10	Alterar proposta de crédito	Dados
Calcular taxas do contrato	11	Calcular taxa de impressão	Lógica
	12	Calcular taxa de entrega	Lógica
	13	Calcular IOF	Lógica
	14	Calcular taxa de alteração do contrato	Lógica
	34	Calcular taxas do contrato_RqN	Lógica
	37	Calcular taxas do contrato	Lógica
	38	Enviar mensagem para Calcular taxas do contrato	Dados
	39	Consolidar mensagens de Calcular taxas do contrato	Dados
	49	Tratar taxas do contrato	Dados
Gerar proposta de	20	Gerar proposta de contrato	Dados

contrato			
Gerar contrato com ajuste	36	Gerar contrato com ajuste	Lógica/Dados
Tratar créditos concedidos	44	Tratar créditos concedidos	Dados
Tratar proposta de contrato	45	Tratar proposta de contrato	Dados
Tratar proposta de crédito	46	Tratar proposta de crédito	Dados
Analisar pedido de crédito	53	Analisar pedido de crédito	Lógica/Dados
Verificar contrato sem ajuste	54	Verificar contrato sem ajuste	Lógica/Dados

Os serviços identificados que devem ser desenvolvidos fisicamente (Tabela 9) foram agrupados. Em termos de desenvolvimento, cada agrupamento representa um projeto para implementação de um web service, enquanto que cada serviço foi implementado como uma operação do web service.

A nomenclatura utilizada neste trabalho para web services e suas operações de serviço considerou os seguintes padrões e diretrizes:

- Foram empregados padrões de nomenclatura da linguagem de programação Java;
- Os projetos de web services que contemplaram serviços candidatos de lógica ou lógica juntamente com dados tiveram o nome de seu web service finalizado com "LogicService". Exemplo: TratarTaxaJurosLogicService;
- Os projetos de web services que contemplaram apenas serviços candidatos de dados tiveram o nome de seu web service finalizado com "DataService". Exemplo: TratarPropostaContratoDataService;
- Serviços identificados a partir da heurística de cluster correspondem a serviços que executam operações CRUD em entidades existentes no modelo canônico. Logo, estes serviços corresponderam a serviços básicos de dados e foram desenvolvidos separadamente por possuírem maior índice de reuso e não conterem lógica de negócio. A Tabela 10 apresenta os serviços que foram desenvolvidos e suas operações.

Tabela 10 - Serviços desenvolvidos e suas operações

Projeto de Serviço	Índice	Operações
VerificarLimiteCreditoLogicService	1	Aprovar crédito
	2	Verificar limite de crédito

TratarTaxaJurosLogicService	3	Valor limite máximo para taxa de juros
	4	Determinar taxa de juros
ManterCadastroClienteLogicService	5	Verificar cliente novo
	6	Verificar cadastro de cliente desatualizado
	52	Manter cadastro do cliente
VerificarContratoRiscoLogicService	8	Ajustar proposta de crédito
	9	Identificar contrato de risco
	10	Alterar proposta de crédito
CalcularTaxasContratoLogicService	11	Calcular taxa de impressão
	12	Calcular taxa de entrega
	13	Calcular IOF
	14	Calcular taxa de alteração do contrato
	34	Calcular taxas do contrato_RqN
	37	Calcular taxas do contrato
	38	Enviar mensagem para Calcular taxas do contrato
	39	Consolidar mensagens de Calcular taxas do contrato
GerarPropostaContratoDataService	20	Gerar proposta de contrato
GerarContratoComAjusteLogicService	36	Gerar contrato com ajuste
TratarCreditosConcedidosDataService	44	Tratar créditos concedidos
TratarPropostaContratoDataService	45	Tratar proposta de contrato
TratarPropostaCreditoDataService	46	Tratar proposta de crédito
TratarClienteDataService	48	Tratar cadastro do cliente
TratarTaxasContratoDataService	49	Tratar taxas do contrato
TratarTaxasJurosDataService	51	Tratar taxas de juros
AnalisarPedidoCreditoLogicService	53	Analisar pedido de crédito
VerificarContratoSemAjusteLogicService	54	Verificar contrato sem ajuste

4.1 Elaboração da interface do serviço.

4.1.1 Desenvolvimento dos esquemas

Os esquemas foram desenvolvidos utilizando o padrão XML Schema – aprovado como especificação pela W3C e incorporado pelo WS-I BP – em sua versão 1.0, utilizando a plataforma Eclipse WTP.

Seguindo os serviços identificados, analisados e projetados por Souza *et al.* [2011], o modelo canônico apresentado por eles foi revisto devido à necessidade de algumas alterações como: inclusão de chaves primárias, definição de direção da associação e inclusão de um atributo no relacionamento entre PropostaCredito e Peca para indicar a quantidade de peças contempladas na proposta de crédito. Este atributo ficou armazenado na classe associativa ItemPeca. O resultado final é apresentado na Figura 31.

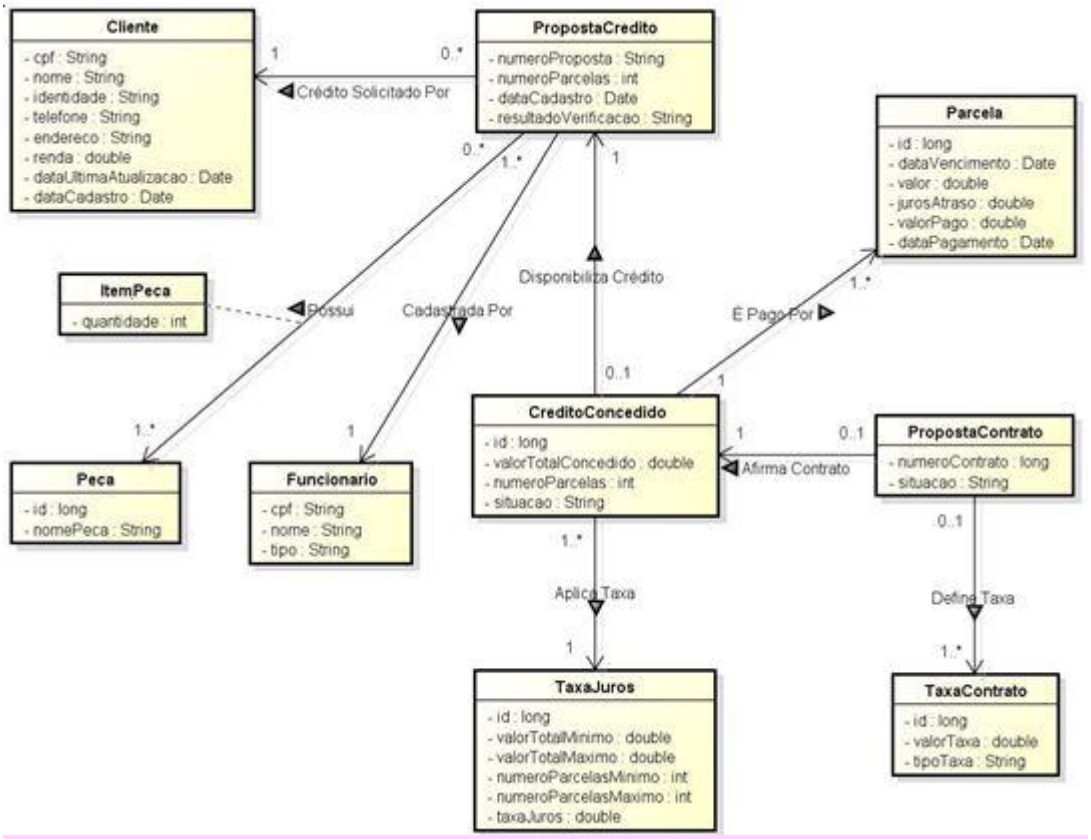

Figura 31 – Modelo canônico do processo de análise de crédito (adaptado de Souza *et al.* [2011])

Seguindo a abordagem de centralização de esquema proposta, foi desenvolvido um esquema central, chamado *ModeloCanonico.xsd*, contendo todas as entidades identificadas no modelo canônico da Figura 31. Este esquema conteve apenas as entidades e seus tipos de dados básicos, ou seja, não incluiu relacionamentos. O objetivo é que os serviços possam reaproveitar os tipos de dados e consigam manter a mesma estrutura do modelo canônico, facilitando a manutenção da estrutura. Já o contrato do serviço se utiliza de esquemas que denominamos esquemas de visão, onde são reaproveitados os tipos complexos do esquema central e são mapeados os relacionamentos das entidades utilizadas no esquema de visão. Isto permite que apenas as estruturas necessárias sejam trafegadas, ou seja, mesmo que o modelo de dados conceitual seja todo interligado, somente os dados necessários ao consumo do serviço são trafegados.

O esquema apresentado na Figura 32 e na Figura 33 pode ser utilizado em um projeto como qualquer outro XSD. Seu objetivo é manter centralizados os tipos de dados referentes ao domínio do processo de análise de crédito, sem considerar os

relacionamentos entre estes tipos de dados. Sua utilização deve ser feita indiretamente através da utilização de outros esquemas, os esquemas de visão - abordagem proposta neste trabalho. Estes esquemas importam o modelo canônico e definem os relacionamentos entre suas entidades (o que não é contemplado no canônico). Assim, se for identificada a necessidade de manipulação ou visualização de dados de qualquer conjunto de entidades no processo de análise de crédito, sempre será utilizado o tipo complexo desta entidade definido no esquema *ModeloCanonico.xsd* e, além disso, será necessário utilizar outros esquemas que fazem os mapeamentos dos relacionamentos entre estas entidades. Por exemplo, para desenvolver um serviço que gera um relatório de todos os créditos concedidos a um determinado cliente, é necessário criar um XSD de visão que utilize os tipos complexos Cliente e CreditoConcedido, importados a partir do esquema central. A Figura 34 mostra um exemplo do esquema representando a visão da entidade PropostaContrato. Observe a visão criada "PropostaContratoView" que representa o relacionamento entre "PropostaContrato", "CreditoConcedido" e "TaxaJuros".

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://uniriotec.monografia.br/ModeloCanonico"
    xmlns:tns="http://uniriotec.monografia.br/ModeloCanonico"
    elementFormDefault="qualified">

    <xsd:annotation>
        <xsd:documentation>
            Definição dos tipos complexos definidos para o processo de
                Análise de Pedido de Crédito.
            Os relacionamentos entre os tipos complexos não estão
                definidos neste XML Schema.
            Os relacionamentos de cada entidade estão representados em outros
                XML Schema que foram chamados de Views.
        </xsd:documentation>
    </xsd:annotation>

    <xsd:complexType name="Cliente">
        <xsd:sequence>
            <xsd:element name="cpf" type="xsd:string" nillable="true"/>
            <xsd:element name="nome" type="xsd:string" nillable="true"/>
            <xsd:element name="identidade" type="xsd:string" nillable="true"/>
            <xsd:element name="telefone" type="xsd:string" />
            <xsd:element name="endereco" type="xsd:string" />
            <xsd:element name="renda" type="xsd:double" />
            <xsd:element name="dataUltimaAtualizacao" type="xsd:date" minOccurs="0" />
            <xsd:element name="dataCadastro" type="xsd:date" minOccurs="0" />
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="PropostaCredito">
        <xsd:sequence>
            <xsd:element name="numeroProposta" type="xsd:string" minOccurs="0"/>
            <xsd:element name="numeroParcelas" type="xsd:int" />
            <xsd:element name="dataCadastro" type="xsd:date" minOccurs="0"/>
            <xsd:element name="resultadoVerificacao" type="xsd:string" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="Peca">
        <xsd:sequence>
            <xsd:element name="id" type="xsd:long" minOccurs="0"/>
            <xsd:element name="nomePeca" type="xsd:string" />
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="ItemPeca">
        <xsd:sequence>
            <xsd:element name="id" type="xsd:long" minOccurs="0"/>
            <xsd:element name="quantidade" type="xsd:int" />
        </xsd:sequence>
    </xsd:complexType>
```

Figura 32 - Modelo Canonico.xsd - parte 1

```
    <xsd:complexType name="Funcionario">
        <xsd:sequence>
            <xsd:element name="cpf" type="xsd:string" />
            <xsd:element name="nome" type="xsd:string" />
            <xsd:element name="tipo" type="xsd:string" />
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="CreditoConcedido">
        <xsd:sequence>
            <xsd:element name="id" type="xsd:long" minOccurs="0"/>
            <xsd:element name="valorTotalConcedido" type="xsd:double" />
            <xsd:element name="numeroParcelas" type="xsd:int" />
            <xsd:element name="situacao" type="xsd:string" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="TaxaJuros">
        <xsd:sequence>
            <xsd:element name="id" type="xsd:long" minOccurs="0"/>
            <xsd:element name="valorTotalMinimo" type="xsd:double" />
            <xsd:element name="valorTotalMaximo" type="xsd:double" />
            <xsd:element name="numeroParcelasMinimo" type="xsd:int" />
            <xsd:element name="numeroParcelasMaximo" type="xsd:int" />
            <xsd:element name="taxaJuros" type="xsd:double" />
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="Parcela">
        <xsd:sequence>
            <xsd:element name="id" type="xsd:long" minOccurs="0"/>
            <xsd:element name="dataVencimento" type="xsd:date" />
            <xsd:element name="valor" type="xsd:double" />
            <xsd:element name="jurosAtraso" type="xsd:double" />
            <xsd:element name="valorPago" type="xsd:double" minOccurs="0"/>
            <xsd:element name="dataPagamento" type="xsd:date" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="PropostaContrato">
        <xsd:sequence>
            <xsd:element name="numeroContrato" type="xsd:long" minOccurs="0"/>
            <xsd:element name="situacao" type="xsd:string" minOccurs="0"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:complexType name="TaxaContrato">
        <xsd:sequence>
            <xsd:element name="id" type="xsd:long" minOccurs="0"/>
            <xsd:element name="valorTaxa" type="xsd:double" />
            <xsd:element name="tipoTaxa" type="xsd:string" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Figura 33 - Modelo Canonico.xsd - parte 2

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://uniriotec.monografia.br/PropostaContratoView"
    xmlns:tns="http://uniriotec.monografia.br/PropostaContratoView"
    elementFormDefault="qualified"
    xmlns:mc="http://uniriotec.monografia.br/ModeloCanonico">

    <xsd:annotation>
        <xsd:documentation>
            Este XML Schema tem a finalidade de representar os relacionamentos da entidade
            "PropostaContrato", com outras entidades identificadas no modelo canônico.
        </xsd:documentation>
    </xsd:annotation>

    <xsd:import namespace="http://uniriotec.monografia.br/ModeloCanonico"
                schemaLocation="../ModeloCanonico.xsd">
    </xsd:import>

    <xsd:complexType name="PropostaContratoView">
        <xsd:sequence>
            <xsd:element name="propostaContrato" type="mc:PropostaContrato"
                minOccurs="1" maxOccurs="1" />
            <xsd:element name="creditoConcedido" type="mc:CreditoConcedido"
                minOccurs="0" maxOccurs="1" />
            <xsd:element name="taxasContrato" type="mc:TaxaContrato"
                minOccurs="1" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

</xsd:schema>
```

Figura 34 - Esquema de visão da entidade PropostaContrato

Esta divisão por visões permite limitar o conteúdo das mensagens SOAP. Assim, o modelo canônico fica conceitualmente interligado, mas, em termos práticos, não é necessário enviar um XML tão extenso em uma mensagem onde o objetivo é obter um simples valor para análise. Por exemplo, para consultar os dados de uma proposta de contrato, apenas é necessário visualizar a proposta de contrato, as taxas de contrato e o crédito concedido da mesma. No entanto, não realizando a divisão por visões, a mensagem XML poderia incluir a proposta de crédito e as taxas de juros relativas ao crédito concedido, e trafegaria também as informações de cliente relativas à proposta de crédito. Dessa forma, a simples consulta incluiria informações de todos os dados da análise de crédito, o que nem sempre é o desejado.

Cada esquema de visão deste projeto representa uma estrutura de dados do domínio e seus relacionamentos definidos. Portanto, foram criados 11 (onze) arquivos XSD: um para o modelo canônico e outros 10 para mapear os relacionamentos. É importante ressaltar que o mapeamento relacional tradicional do modelo canônico, onde uma entidade possui uma chave estrangeira para outra entidade a qual está relacionada, não está presente em cada esquema de visão. O

relacionamento será realizado em uma camada de acesso a dados objeto-relacional, que será abordada mais à frente neste trabalho.

É possível observar nos esquemas (Figura 32, Figura 33 e Figura 34) o uso de restrições nos elementos, por exemplo:

- A obrigatoriedade do preenchimento da data de vencimento de uma parcela;
- O valor mínimo de 1(um) para o número de parcelas de um crédito concedido;
- A obrigatoriedade de pelo menos uma taxa de contrato associada a uma proposta de contrato.

Estas restrições permitem que a mensagem possa ser validada, garantindo que estas regras de negócio sejam atendidas, antes que seja criada uma instância do serviço, evitando processamentos desnecessários.

4.1.2 Desenvolvimento do WSDL

Conforme descrito na seção 2.3, foi utilizada a versão 1.1 para o desenvolvimento dos contratos dos serviços, através também da plataforma Eclipse WTP. Os WSDLs foram desenvolvidos com as informações presentes na Tabela 10, uma vez que esta disponibiliza todas as funcionalidades dos serviços. No entanto, as informações de entrada e saída não foram mapeadas e, por isso, foi necessário revisitar o processo de análise de crédito para identificar quais informações eram necessárias em cada operação. Em seguida, essas informações foram comparadas com as informações das entidades mapeadas no modelo canônico. Desta forma, as entradas e saídas que representam entidades do modelo canônico foram definidas no seu devido esquema de visão.

O desenvolvimento do WSDL foi realizado manualmente e teve como base as exigências do Basic Profile da WS-I, apresentadas na seção 3.2, e o conceito de contract-first e suas possibilidades, como modularização e a reutilização dos tipos de dados desenvolvidos na seção 4.1.1. Como exemplo, a Figura 35 apresenta o WSDL abstrato do serviço Tratar Cliente. O WSDL apresentado contém apenas as definições do serviço, as operações, as mensagens e os tipos de dados relacionados. Algumas observações sobre este WSDL são apresentadas a seguir. Na Figura 35

foram incluídos os números referentes a cada observação para facilitar a ilustração do que é apresentado.

1. Definição do nome do serviço seguindo o padrão: Nome do serviço identificado + Tipo de serviço (Dados ou Lógica) + *Service*.
2. Definição do *namespace* do serviço seguindo o padrão: http://uniriotec.monografia.br/ + nome do serviço + tipo de WSDL (*defs* para abstrato ou *service* para concreto).
3. Importação de um esquema de visão, descrito na seção 4.1.1.
4. Definição da estrutura de mensagem seguindo o padrão: nome da operação + tipo de mensagem (*Request* ou *Response*).
5. Reutilização de tipos complexos definidos nos esquemas de visão para ser utilizado como parte da mensagem.
6. Definição do *port type* seguindo o padrão de nomenclatura do item 1.
7. Definição do nome da operação:
    a. Serviços identificados a partir da heurística de cluster seguem o seguinte padrão para operações CRUD: incluir, consultar, atualizar ou excluir + nome da entidade;
    b. Demais serviços: utilizam a nomenclatura das operações presentes na Tabela 10. Entretanto, as funções foram renomeadas para seguir o padrão Java de nomenclatura. Operações são iniciadas com letra minúscula. A partir daí, cada palavra é iniciada por letra maiúscula. Preposições e caracteres especiais foram removidos.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions name="TratarClienteDataService" 1
    targetNamespace="http://uniriotec.monografia.br/TratarClienteDataService/defs" 2
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://uniriotec.monografia.br/TratarClienteDataService/defs"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:cv="http://uniriotec.monografia.br/ClienteView">

    <wsdl:documentation>
        WSDL contendo as definições abstratas do serviço. Essas definições indicam o que é feito pelo serviço,
        as operações disponíveis e tipos de dados utilizados nas operações.
    </wsdl:documentation>

    <wsdl:types>
        <xsd:schema>
            <xsd:import namespace="http://uniriotec.monografia.br/ClienteView"
                schemaLocation="http://localhost:8080/AnaliseCredito/visao/ClienteView.xsd"> 3
            </xsd:import>
        </xsd:schema>
    </wsdl:types>

    <wsdl:message name="incluirClienteRequest"> 4
        <wsdl:part name="clienteView" type="cv:ClienteView" /> 5
    </wsdl:message>
    <wsdl:message name="incluirClienteResponse">
        <wsdl:part name="sucesso" type="xsd:boolean" />
    </wsdl:message>

    <wsdl:message name="consultarClienteRequest">
        <wsdl:part name="cpf" type="xsd:string" />
    </wsdl:message>
    <wsdl:message name="consultarClienteResponse">
        <wsdl:part name="clienteView" type="cv:ClienteView" />
    </wsdl:message>

    <wsdl:message name="excluirClienteRequest">
        <wsdl:part name="cpf" type="xsd:string" />
    </wsdl:message>
    <wsdl:message name="excluirClienteResponse">
        <wsdl:part name="sucesso" type="xsd:boolean" />
    </wsdl:message>

    <wsdl:message name="alterarClienteRequest">
        <wsdl:part name="clienteView" type="cv:ClienteView" />
    </wsdl:message>
    <wsdl:message name="alterarClienteResponse">
        <wsdl:part name="sucesso" type="xsd:boolean" />
    </wsdl:message>

    <wsdl:portType name="TratarClienteDataService"> 6
        <wsdl:operation name="incluirCliente"> 7
            <wsdl:input message="tns:incluirClienteRequest" />
            <wsdl:output message="tns:incluirClienteResponse" />
        </wsdl:operation>

        <wsdl:operation name="consultarCliente">
            <wsdl:input message="tns:consultarClienteRequest" />
            <wsdl:output message="tns:consultarClienteResponse" />
        </wsdl:operation>

        <wsdl:operation name="excluirCliente">
            <wsdl:input message="tns:excluirClienteRequest" />
            <wsdl:output message="tns:excluirClienteResponse" />
        </wsdl:operation>

        <wsdl:operation name="alterarCliente">
            <wsdl:input message="tns:alterarClienteRequest" />
            <wsdl:output message="tns:alterarClienteResponse" />
        </wsdl:operation>
    </wsdl:portType>

</wsdl:definitions>
```

Figura 35 - WSDL abstrato de Tratar Cliente

Após a definição da interface abstrata do serviço, deve ser desenvolvido o WSDL concreto do serviço, definindo detalhes de implementação. Este WSDL será o

utilizado pelos consumidores do serviço. A Figura 36 apresenta o WSDL concreto do serviço “Tratar Cliente”. Os principais itens a serem observados no WSDL são:

1. Definição do *namespace* do serviço seguindo o padrão: http://uniriotec.monografia.br/ + nome do serviço + *service* (*defs* para abstrato ou *service* para concreto);
2. Importação do WSDL abstrato para que sejam especificados detalhes de implementação das operações e serviço definidos anteriormente;
3. Seção para definição dos atributos de detalhes de implementação relativos à formatação (*literal* ou *encode*) e estilo (*document* ou *rpc*) de mensagem do serviço “TratarClienteDataService”;
4. Definição do padrão RPC (*Remote Procedure Call*) para *binding* do serviço. Este estilo foi utilizado por ser simples e intuitivo para o desenvolvimento e estar de acordo com o WS-I BP;
5. Definição do protocolo SOAP sobre HTTP para envio das mensagens;
6. Uso de *literal* para troca de mensagens. O uso de *encoded* não está de acordo com o WS-I;
7. Definição do *endpoint* do serviço;
8. Especificação do *endpoint* em sua URL de acesso.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<wsdl:definitions name="TratarClienteDataService"
    targetNamespace="http://uniriotec.monografia.br/TratarClienteDataService/service"1
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://uniriotec.monografia.br/TratarClienteDataService/service"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:defs="http://uniriotec.monografia.br/TratarClienteDataService/defs">

    <wsdl:documentation>
        O WSDL concreto define como e onde consumir este serviço.
        Esse padrão de projeto permite fornecer diferentes possibilidades
        de binding e endpoints mantendo as definições abstratas do serviço.
    </wsdl:documentation>

    <wsdl:import namespace="http://uniriotec.monografia.br/TratarClienteDataService/defs"
            location="TratarClienteDataServiceAbstract.wsdl" /> 2

    <wsdl:binding name="TratarClienteDataServicePort" type="defs:TratarClienteDataService"> 3
        <soap:binding style="rpc" 4
            transport="http://schemas.xmlsoap.org/soap/http" /> 5
        <wsdl:operation name="incluirCliente">
            <soap:operation soapAction="http://uniriotec.monografia.br/TratarClienteDataService/incluirCliente" />
            <wsdl:input>
                <soap:body use="literal" namespace="http://uniriotec.monografia.br/TratarClienteDataService/" /> 6
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal" namespace="http://uniriotec.monografia.br/TratarClienteDataService/" />
            </wsdl:output>
        </wsdl:operation>

        <wsdl:operation name="consultarCliente">
            <soap:operation soapAction="http://uniriotec.monografia.br/TratarClienteDataService/consultarCliente" />
            <wsdl:input>
                <soap:body use="literal" namespace="http://uniriotec.monografia.br/TratarClienteDataService/"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal" namespace="http://uniriotec.monografia.br/TratarClienteDataService/"/>
            </wsdl:output>
        </wsdl:operation>

        <wsdl:operation name="excluirCliente">
            <soap:operation soapAction="http://uniriotec.monografia.br/TratarClienteDataService/excluirCliente" />
            <wsdl:input>
                <soap:body use="literal"  namespace="http://uniriotec.monografia.br/TratarClienteDataService/"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal"  namespace="http://uniriotec.monografia.br/TratarClienteDataService/"/>
            </wsdl:output>
        </wsdl:operation>

        <wsdl:operation name="alterarCliente">
            <soap:operation soapAction="http://uniriotec.monografia.br/TratarClienteDataService/alterarCliente" />
            <wsdl:input>
                <soap:body use="literal" namespace="http://uniriotec.monografia.br/TratarClienteDataService/"/>
            </wsdl:input>
            <wsdl:output>
                <soap:body use="literal" namespace="http://uniriotec.monografia.br/TratarClienteDataService/"/>
            </wsdl:output>
        </wsdl:operation>
    </wsdl:binding>

    <wsdl:service name="TratarClienteDataService"> 7
        <wsdl:port binding="tns:TratarClienteDataServicePort" name="TratarClienteDataServicePort">
            <soap:address
                location="http://soaservices.servehttp.com:8080/TratarCliente/TratarClienteDataService" /> 8
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Figura 36 - WSDL Concreto do serviço Tratar Cliente

Após o fim do desenvolvimento do WSDL, é importante certificar que o mesmo atende aos padrões de interoperabilidade definidos pela WS-I. Para isso, foi utilizada a ferramenta SOAP UI[2] para realizar testes de interoperabilidade em um

[2] http://www.soapui.org/

WSDL. Esta mesma ferramenta será abordada mais a frente para testes funcionais de web services.

O SOAPUI realiza a verificação de todas as normas apresentadas pelo WS-I BP, informa o resultado para cada item e apresenta um sumário da verificação do WSDL em geral[3]. Para cada norma de interoperabilidade, o relatório informa o resultado do teste (aprovado, reprovado, aviso, não aplicável, faltando insumo), apresentando uma mensagem detalhada das não conformidades, se estas existirem. A Figura 37 apresenta um exemplo de uso da ferramenta SOAPUI para validar o serviço "TratarClienteDataService", cujo WSDL foi apresentado na Figura 35 e Figura 36.

Summary

Result passed

Artifact Targets Analyzed: The summary result applies to the following artifact targets which were specified in the analyzer configuration file.

Description binding=TratarClienteDataServicePort

Assertion Result Summary:

Assertion ID	Passed	Failed	Prerequsite Failed	Warning	Not Applicable	Missing Input
BP2010	1	0	0	0	0	
BP2011	1	0	0	0	0	
BP2012	0	0	0	0	1	
BP2013	1	0	0	0	0	
BP2014	0	0	0	0	4	
BP2017	1	0	0	0	0	
BP2018	2	0	0	0	0	
BP2019	0	0	0	0	1	
BP2020	1	0	0	0	0	
BP2021	0	0	0	0	1	

Figura 37 - Sumário de verificação WS-I

4.2 Desenvolvimento de serviço

O desenvolvimento foi dividido em duas etapas. Antes da real implementação dos serviços, foi projetada e desenvolvida a estrutura de transferência (mapeamento) de dados e a camada de acesso e persistência dos dados em um SGBD.

Para o desenvolvimento, houve uma preocupação em utilizar os principais padrões e recursos mais atuais da linguagem Java, em especial a especificação JEE6[4]. Isto propicia maior facilidade de suporte e manutenção dos frameworks

[3] http://www.soapui.org/About-SoapUI/features.html#tech-support
[4] http://jcp.org/en/jsr/detail?id=316

utilizados, uma vez que os mesmos são mantidos pelas principais empresas do mercado em conjunto com a mantenedora da linguagem Java, a Oracle.

4.2.1 Mapeamento Schema x Java

Em se tratando de web services, os dados a serem trafegados são mensagens SOAP e contém estruturas de dados como partes das mensagens. Para que um serviço desenvolvido em Java possa tratar corretamente essas mensagens, deve haver algum tipo de ligação (*binding*) entre dados em XML e classes Java [Metha, 2003]. Para realizar esta ligação foi utilizado o JAXB (*Java Architecture for XML Binding*)[5]. JAXB é a especificação oficial JEE6 para relacionar estruturas de dados Java e XML. Permite mapear as classes Java através de anotações, correlacionando tipos e/ou elementos de um XML Schema e atributos de classe POJO (*Plain Old Java Object*). Sua versão atual (2.0) é mantida por profissionais da Oracle (Sun) e por grupos das principais empresas do mercado em SOA.

O framework trabalha a partir da serialização de documentos XML em classes JAXB (*UnMarshalling*) para mapear dados para o serviço. Para isso, é necessário que o pacote de classes contenha uma fábrica de objetos, com o dever de inicializar cada objeto serializado. Ao mapear dados do serviço para mensagens SOAP, o framework serializa instâncias em formato XML (*Marshalling*).

Com o domínio do processo bem definido e centralizado, é possível manter as classes JAXB em um único projeto que pode ser utilizado por vários serviços.

A Figura 38 apresenta o comparativo entre uma classe JAXB e um arquivo XML Schema, representando a visão da entidade CreditoConcedido. Cada item está correlacionado entre a classe Java e o XSD pelos números em vermelho. A partir deste mapeamento, um serviço é capaz de capturar dados recebidos e enviar dados sem que haja perda de informações ou incompatibilidade de dados.

[5] http://jaxb.java.net/

Figura 38 - Correlação JAXB x XSD

Neste trabalho foram abordadas as seguintes anotações JAXB. Dentre elas, apenas a *XmlTransient* não foi utilizada no exemplo apresentado na Figura 38.

- *XmlAccessorType* – Indica se os elementos serão acessados pelos atributos da classe (*Field*) ou através de funções *get* (*Property*);
- *XmlType* – Relaciona um *complexType* ou *simpleType* de um XSD com uma classe Java. Deve ser definido o *namespace* e o nome do tipo complexo como

parâmetros desta anotação. Além disso, o parâmetro *propOrder* deve ser especificado para identificar a ordem e quais elementos devem estar em um *bind* para XML;

- *XmlElement* – Relaciona um *element* de um XSD com um atributo de objeto Java. Devem ser definidos o namespace e o nome do elemento como parâmetros desta anotação. Além disso, deve ser observada a relação do parâmetro *maxOccurs*. Se este parâmetro estiver configurado para receber mais de um elemento, este deve ser definido com uma lista na classe POJO;
- *XmlTransient* – Indica que um determinado elemento não será capturado na serialização de objetos. Isto permite que campos auxiliares ou *flags* possam ser utilizados na classe JAXB sem que o framework utilize estes elementos.

Para que os objetos JAXB pudessem representar um tipo complexo de visão, o padrão DTO (*Data Transfer Object*) foi utilizado. Este padrão habilita que uma série de conjuntos de dados possa ser trafegado em um único objeto, evitando envio de série de parâmetros ou uso de múltiplas chamadas. Fowler [2002] aborda o uso do padrão DTO para serialização automática de dados entre XML e Java, utilizando métodos de transformação. No entanto, com JAXB esta serialização se torna totalmente automatizada através das anotações.

Os objetos que representam os esquemas de visões utilizam o padrão DTO para encapsular a entidade que dá nome à visão e às entidades que representam seus relacionamentos. Desta forma, o objeto CreditoConcedidoView da Figura 38 representa o tipo complexo CreditoConcedidoView, encapsulando os elementos presentes no esquema, que são: CreditoConcedido, PropostaCredito, TaxaJuros e Parcela.

4.2.2 Persistência de dados

Assim como em softwares de arquitetura tradicionais, serviços baseados em SOA devem possuir uma camada de persistência dos dados trafegados entre um SGBD e o usuário final. Neste trabalho, foi utilizado o framework JPA (*Java*

Persistence API)[6], presente na especificação JEE6, que define um mapeamento objeto-relacional das entidades presentes no banco de dados para objetos Java simples [Burke e Monson-Haefel, 2007]. Este mapeamento aproxima o banco de dados do desenvolvimento, uma vez que as aplicações são desenvolvidas em uma programação orientada a objetos.

O framework trabalha com o uso de anotações, permitindo que cada classe Java possa ser mapeada para uma tabela do banco de dados. Todos os aspectos relativos à tabela podem ser trazidos à entidade JPA como: colunas, chave primária, relacionamentos, restrições de unicidade, definição de sequências, gatilhos (*triggers*), entre outros.

Desta forma, realizar operações básicas (CRUD) podem ser feitas com pouca escrita de código pelo desenvolvedor. Além disso, a JPA possui uma linguagem inspirada em SQL para consultas avançadas voltada para as entidades ao invés de tabelas, mais familiar à linguagem Java, chamada JPQL (*Java Persistence Query Language*).

Para cada entidade do modelo canônico do processo apresentado na Figura 31, uma classe anotada como entidade foi criada em um projeto representando todo o modelo. A Figura 39 apresenta a entidade CreditoConcedido, permitindo visualizar algumas das principais anotações JPA. A seguir, o detalhamento destas anotações [Burke e Monson-Haefel, 2007]:

- ***Entity*** – Define que a classe Java é uma entidade JPA;
- ***Table*** – Relaciona a entidade mapeada com uma determinada tabela do banco de dados, através do parâmetro *name*, que deve ser igual ao nome da tabela;
- ***DynamicUpdate*** – Com o valor *true*, a entidade persistida só irá alterar os valores que não estiverem nulos;
- ***Id*** – Identifica o atributo da classe como a chave primária da tabela;
- ***SequenceGenerator*** – Define uma *sequence* no SGBD e atribui o valor sequencial ao atributo que está sendo mapeado. Devem ser definidos o nome da sequência e o valor incremental (*allocationSize*).

[6] http://www.jcp.org/en/jsr/detail?id=317

Esta anotação não é independente de plataforma, visto que existem sistemas que não possuem gerador de sequência;

- ***GeneratedValue*** – Complementar ao *SequenceGenerator*, define o tipo valor automático do atributo, que pode ser por sequência, tabela, identidade ou mesmo escolha automática. Esta escolha deve variar de acordo com a implementação do SGBD;
- ***Column*** – Mapeia diretamente um atributo do objeto a uma coluna da tabela. Caso o nome seja diferente, o atributo *name* deve ser utilizado para manter essa relação. Além disso, outras propriedades da coluna podem ser definidas como: restrição de unicidade, permissão de valores nulos, tamanho permitido, etc;
- ***JoinColumn*** – Mapeia um atributo da classe como uma coluna da tabela. No entanto, esta coluna é definida como uma chave estrangeira para outra tabela. Caso o nome seja diferente, o atributo *name* deve ser utilizado para manter essa correlação.
- ***OneToOne*** – Define um relacionamento do tipo 1..1;
- ***ManyToOne*** – Define um relacionamento do tipo N..1;
- ***OneToMany*** – Define um relacionamento do tipo 1..N. Neste caso, a chave estrangeira deve estar presente na tabela relacionada. Portanto, o atributo *name* da anotação *JoinColumn* deve especificar a coluna da tabela relacionada.

```java
package br.uniriotec.monografia.entidades;

import java.io.Serializable;

/*
 * Annotations de JPA
 */
@Entity
@Table(name="creditos_concedidos")
@DynamicUpdate(value=true)
/*
 * Annotations de JAXB
 */
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "CreditoConcedido", propOrder = {
    "id",
    "valorTotalConcedido",
    "numeroParcelas",
    "situacao"
})
public class CreditoConcedido implements Serializable{

    private static final long serialVersionUID = 1L;

    @Id
    @SequenceGenerator(name = "CREDITO_CONCEDIDO_ID",
            sequenceName = "CRED_CONC_SEQ", allocationSize = 1)
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    @Column(name = "id")
    @XmlElement(required = false)
    private long id;

    @Column(name = "valor_total_concedido")
    @XmlElement(required = true)
    private double valorTotalConcedido;

    @Column(name = "numero_parcelas")
    @XmlElement(required = true)
    private int numeroParcelas;

    @Column(name = "situacao")
    @XmlElement(required = false)
    private String situacao;

    @OneToOne(cascade=CascadeType.ALL)
    @JoinColumn(name="numero_proposta_credito")
    @XmlTransient
    private PropostaCredito propostaCredito;

    @ManyToOne(cascade=CascadeType.ALL)
    @JoinColumn(name="id_taxa_juros")
    @XmlTransient
    private TaxaJuros taxaJuros;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="id_credito_concedido")
    @XmlTransient
    private List<Parcela> parcelas;
```

Figura 39 - Entidade CreditoConcedido com as anotações JPA

A especificação JPA é vasta e tem o objetivo de incorporar às classes Java todas as possibilidades existentes em um banco de dados relacional, existindo inclusive algumas ferramentas para geração automática da estrutura do banco de dados. A Tabela 11 apresenta algumas outras possibilidades de mapeamento do framework, e algumas delas que também foram utilizadas neste trabalho.

Tabela 11 - Listagem de anotações JPA

Anotação	Descrição
IdClass	Define uma classe para ser chave primária de uma tabela. Isto indica que esta chave será composta.
TableGenerator	Semelhante à anotação *SequenceGenerator*. No entanto, define uma tabela geradora de valores sequenciais. Ao contrario do gerador de sequência, é permitido em todos os SGBDs.
Transient	Identifica um atributo do objeto como **não** gerenciável pela unidade de persistência.
Temporal	Utilizado para definir tipos de dados em formato data. Pode ser parametrizado para diferentes formatos de data, como data/hora.
Lob	Identifica um atributo da classe como uma coluna especial do tipo *Lob*, que permite armazenamento de grande volume de dados.
ManyToMany	Define um relacionamento do tipo N × M
Pre(Persist, Update, Remove)	Aplicação similar a uma trigger executada antes de inserir/atualizar/excluir dados da tabela. Esta anotação é utilizada em uma função.
Post(Load,Persist, Update, Remove)	Aplicação similar a uma trigger executada após carregar/inserir/atualizar/excluir dados da tabela. Esta anotação é utilizada em uma função.

Na Figura 39 podem também ser observadas a presença de anotações do framework JAXB. Conforme abordado por Burkee e Monson-Haefel [2007], uma das facilidades da especificação JPA é que, pelo fato dos beans de entidade serem mapeados em objetos Java simples (POJO), essas entidades podem ser utilizadas para transferir dados entre cliente e servidor.

Como as estruturas de dados que devem representar os tipos de dados definidos pela Figura 32 e pela Figura 33 são bastante semelhantes com as entidades mapeadas, estas classes foram utilizadas tanto para persistir dados quanto para transferência de dados das mensagens SOAP.

Sendo assim, uma classe possui tanto anotações JAXB quanto anotações JPA. Esta abordagem auxilia na manipulação dos dados, dispensando as conversões trabalhosas de classes que representam a transferência de dados entre consumidor e provedor (JAXB) para classes que representam a persintência de dados (JPA). Além

disso, facilita a manutenção do código, uma vez que os dados estão centralizados. Possíveis alterações dos tipos definidos no XSD refletirão somente neste ponto do projeto.

4.2.3 Camada de Acesso a Dados

Na seção 4.2.2, foi abordado o uso do framework JPA para representação objeto-relacional do banco de dados. No entanto, apenas o uso das entidades levaria aos serviços a necessidade de implementar sua própria lógica de tratamento dos dados.

Para evitar este acoplamento, foi utilizado o conceito de DAO (Data Access Object) apresentado pela Sun e estendido em uma abordagem proposta por Mellqvist [2006] para criação de DAOs genéricos, adotando a funcionalidade Java *Generics*. Utilizando este padrão, é possível que as operações de acesso a dados mais comuns estejam implementadas em apenas uma classe e os demais DAOs gerados estendam esta classe. A Figura 40 e a Figura 41 apresentam o DAO genérico e uma implementação do mesmo, respectivamente.

```java
public class EntidadeDAO<T> implements IEntidadeDAO<T> {

    @PersistenceContext(unitName = "ModeloCanonico")
    private EntityManager em;
    private Class<?> classe;

    public EntidadeDAO(Class<?> classe) {
        this.classe = classe;
    }

    @Override
    public void gravar(T entidade) {
        try {
            em.persist(entidade);
            em.flush();
        } catch (Exception e) {
            Log.getLog().error("Erro ao inserir. ", e);
        }
    }

    @Override
    @SuppressWarnings("unchecked")
    public T carregar(Object pkEntidade) {
        Object entidade = new Object();
        entidade = em.find(classe, pkEntidade);
        em.flush();
        return (T) entidade;
    }

    @Override
    public void atualizar(T entidade) {
        em.merge(entidade);
        em.flush();
    }

    @Override
    public void excluir(T entidade) {
        em.remove(entidade);
        em.flush();
    }

    @Override
    @SuppressWarnings("unchecked")
    public List<T> carregarTodos() {
        Session sessao = (Session) em.getDelegate();
        Criteria criteria = sessao.createCriteria(classe);
        return criteria.list();

    }

    @SuppressWarnings("unchecked")
    public List<T> carregarComRestricoes(Map<String, Object> parametros) {
        Session sessao = (Session) em.getDelegate();
        Criteria criteria = sessao.createCriteria(classe);

        criteria.add(Restrictions.allEq(parametros));

        return criteria.list();
    }
}
```

Figura 40 - DAO Genérico

```java
package br.uniriotec.monografia.dao.implementacoes;

import java.util.HashMap;

@Stateless
public class PropostaContratoDAO extends EntidadeDAO<PropostaContrato> implements IPropostaContratoDAO{

    public PropostaContratoDAO(Class<?> classe) {
        super(classe);
    }

    public PropostaContratoDAO(){
        super(PropostaContrato.class);
    }

    @Override
    public List<PropostaContrato> listarPropostaContratoPorSituacao (String situacao)
    throws IllegalStateException,Exception{
        Map<String, Object> criterios = new HashMap<String, Object>();
        criterios.put("situacao", situacao);

        return carregarComRestricoes(criterios);
    }
```

Figura 41 - DAO da entidade PropostaContrato

É importante observar como a implementação de um DAO a partir da extensão da classe EntidadeDAO possibilita uma economia significativa de código e trabalho repetitivo por este já possuir operações CRUD genéricas prontas. Desta forma, será necessário o desenvolvimento somente de funcionalidades específicas não contempladas no DAO genérico. Além disso, pode-se perceber a existência da anotação *@Stateless* (primeira linha da Figura 41), responsável por tornar o objeto em um EJB (*Enterprise Java Beans*) sem informações de estado. Além disso, a interface do DAO possui a anotação *@Remote*, que permite que o EJB seja utilizado remotamente, ou seja, não é necessário que os DAOs estejam no mesmo ambiente dos serviços.

Conforme abordado por Burkee e Monson-Haefel [2007], EJBs são componentes do lado servidor escalonáveis, transacionais e distribuíveis. Estas características podem ser importantes quando se trata de SOA, onde os serviços podem estar distribuídos em diversos servidores. Além disso, sua utilização possibilita o gerenciamento centralizado dos dados corporativos, cabendo aos serviços apenas a utilização desses EJBs.

Aplicar as capacidades EJB aos DAOs permite adicionar algumas características chave para o desenvolvimento dos serviços deste trabalho:

- Permitir que os DAOs sejam gerenciados pelo servidor de aplicação e possam ser injetados diretamente nos serviços, adicionando tratamento automático de transações e concorrência;

- Injeção automática de contexto de persistência (*EntityManager*), a partir de fonte de dados provida pelo servidor de aplicação, eliminando necessidade de criação de fábrica de conexões. Isto é realizado através da anotação *@PersistenceContext*, informando qual unidade de persistência configurada, conforme apresentado na Figura 40.

Os DAOs foram divididos em dois projetos, de acordo com a semântica do modelo canônico, com o objetivo de minimizar a possibilidade de afetar os serviços em produção, caso parte dos EJBs estejam fora do ar devido à manutenção ou problemas. São eles:

- PropostaCliente – Conjunto de entidades referentes à proposta de crédito para o cliente;
- ConcessaoCredito – Conjunto de entidades referentes à concessão do crédito e oferta de contrato ao cliente.

A Figura 42 apresenta o arquivo persistence.xml do projeto PropostaClienteEJB, responsável por fazer a configuração das conexões gerenciadas pelos EJBs e o pool de conexões provido pelo servidor de aplicação.

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
    version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">
    <persistence-unit name="ModeloCanonico" transaction-type="JTA">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <jta-data-source>jdbc/modelo</jta-data-source>
        <jar-file>[GLASSFISH_HOME]/domains/analise_credito/lib/ext/ModeloCanonico.jar</jar-file>

        <class>br.uniriotec.monografia.entidades.Cliente</class>
        <class>br.uniriotec.monografia.entidades.CreditoConcedido</class>
        <class>br.uniriotec.monografia.entidades.ItemPeca</class>
        <class>br.uniriotec.monografia.entidades.Funcionario</class>
        <class>br.uniriotec.monografia.entidades.Parcela</class>
        <class>br.uniriotec.monografia.entidades.Peca</class>
        <class>br.uniriotec.monografia.entidades.PropostaContrato</class>
        <class>br.uniriotec.monografia.entidades.PropostaCredito</class>
        <class>br.uniriotec.monografia.entidades.TaxaContrato</class>
        <class>br.uniriotec.monografia.entidades.TaxaJuros</class>

        <properties>
            <property name="eclipselink.ddl-generation" value="create-tables" />
            <property name="eclipselink.ddl-generation.output-mode" value="database" />
            <property name="eclipselink.target-database" value="PostgreSQL" />
            <property name="eclipselink.logging.level" value="FINER"/>
        </properties>
    </persistence-unit>
</persistence>
```

Figura 42 - Persistence.xml do projeto PropostaClienteEJB

4.2.4 Implementação do Serviço

Para implementação do contrato do serviço, foi escolhido o framework JAX-WS (também chamado de JWS – Java Web Services) presente na especificação JEE6. Antes, em sua 1ª versão, o framework era chamado de JAX-RPC, e teve mudança na nomenclatura por incorporar maiores funcionalidades e integração direta com JAXB [Kalin, 2009]. O framework é capaz de incorporar integralmente, através do uso de anotações, todas as capacidades de um WSDL 1.1 a uma interface Java.

O uso do JAX-WS se torna fundamental neste trabalho, uma vez que permite utilizar um WSDL já elaborado, tornando possível aplicar a abordagem contract-first para desenvolver os serviços, e obter as vantagens observadas na seção 3.6.

Além disso, optou-se por desenvolver manualmente uma interface do serviço e a implementação do mesmo, com o intuito de evitar geração automática de classes e obter domínio tanto do contrato (WSDL) quanto da implementação do serviço. A ligação entre ambas as classes é feita pelo arquivo de configuração *sun-jaxws.xml*, conforme Figura 43. Este arquivo é capaz de configurar diversos *endpoints* para o serviço, ou seja, configurar várias implementações para um mesmo serviço diferenciando a *url-pattern* para acesso de cada uma das implementações. A possibilidade de ter implementações distintas de um mesmo serviço demonstra a importância do desenvolvimento utilizando contract-first em conjunto com o padrão de projeto de modularização do WSDL, conforme apresentada na seção 3.5.

```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints
  xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime"
  version="2.0">
  <endpoint
      name="TratarCliente"
      implementation="br.uniriotec.monografia.servicos.TratarClienteImpl"
      url-pattern="/TratarClienteDataService"/>
</endpoints>
```

Figura 43 - Arquivo de configuração sun-jaxws.xml

A seguir, serão apresentadas as anotações do JAX-WS utilizadas neste trabalho e sua respectiva aplicação aos serviços desenvolvidos. É importante ressaltar que toda vez que uma anotação ou propriedade da anotação faz referência a alguma

definição contida no WSDL, a mesma deve estar idêntica. Caso contrário, pode causar erro de despacho em uma chamada pelo consumidor ou até mesmo fazer com que o framework ignore o WSDL elaborado e gere automaticamente um novo, entrando em desacordo com a abordagem contract-first.

Web Service

A anotação *@WebService* é usada para especificar que a classe é um serviço ou que a interface define um serviço. As seguintes propriedades podem ser definidas:

- *endpointInterface* – Nome completo da interface Java que define o serviço. Deve seguir o padrão Java de: pacote + nome da classe.
- *name* – Nome que representa a interface do serviço. Deve ser o mesmo definido na tag *portType* do WSDL abstrato.
- *targetNamespace* – *targetNamespace* do serviço e de seus elementos. Deve ser o mesmo definido na tag *definitions* do WSDL concreto.
- *serviceName* – Nome do *serviceName* do serviço. Deve ser o mesmo definido na tag *service* do WSDL concreto.
- *portName* – Nome do *binding* do serviço. Deve ser o mesmo definido na tag *port* do WSDL concreto.
- wsdlLocation – Caminho relativo do WSDL. É **obrigatório** para o framework que o arquivo esteja disponível no contexto para o container web e contido na pasta "wsdl", ou seja, deve estar no formato "WEB-INF/wsdl/nome do arquivo.wsdl"

A Figura 44 e a Figura 45 apresentam o uso da anotação na interface e na implementação do serviço. As propriedades são definidas somente na classe que implementa o serviço.

```
@WebService
public interface TratarCliente {
```

Figura 44 - Anotação @WebService usada na interface do serviço

```
@WebService(
        endpointInterface = "br.uniriotec.monografia.servicos.TratarCliente",
        name = "TratarClienteDataService",
        targetNamespace = "http://uniriotec.monografia.br/TratarClienteDataService/service",
        serviceName = "TratarClienteDataService",
        portName = "TratarClienteDataServicePort",
        wsdlLocation = "WEB-INF/wsdl/TratarClienteDataServiceConcreto.wsdl"
)
public class TratarClienteImpl implements TratarCliente{
```

Figura 45 - Anotação @WebService usada na classe do serviço

SoapBinding

A anotação *@SOAPBinding* (Figura 46) é usada para definir o estilo de comunicação do serviço. É uma referência a tag *soap:binding* definida no WSDL concreto dos serviços. Neste trabalho, foi utilizado somente o estilo RPC-Literal. As seguintes propriedades podem ser definidas:

- *style* – Estilo de *binding* do serviço. Valores possíveis são: RPC ou DOCUMENT. O padrão é *document*;
- *use* – Estilo de codificação das mensagens do serviço. Valores possíveis são: ENCODED ou LITERAL. O padrão é *literal*;
- *parameterStyle* – Define se os dados das mensagens virão em forma de objetos ou em vários parâmetros. Valores possíveis são: BARE ou WRAPPED, respectivamente. O padrão é *wrapped*.

```
@WebService(
        endpointInterface = "br.uniriotec.monografia.servicos.TratarCliente",
        name = "TratarClienteDataService",
        targetNamespace = "http://uniriotec.monografia.br/TratarClienteDataService/service",
        serviceName = "TratarClienteDataService",
        portName = "TratarClienteDataServicePort",
        wsdlLocation = "WEB-INF/wsdl/TratarClienteDataServiceConcreto.wsdl"
)
@SOAPBinding(style = Style.RPC, use = Use.LITERAL)
public class TratarClienteImpl implements TratarCliente{
```

Figura 46 - Anotação @SOAPBinding

WebMethod

A anotação *@WebMethod* (Figura 47) expõe um método da classe como uma operação do serviço. As seguintes propriedades podem ser definidas:

- *operationName* – Nome da operação. Deve ser o mesmo definido na tag *operation* do WSDL abstrato;

- *action* – Nome da ação SOAP. Deve ser o mesmo definido na tag *operation* do WSDL concreto. Caso necessário, deve ser utilizado somente na implementação da operação, não na interface. Alguns servidores de aplicação possuem métodos para validação da ação SOAP chamada, e para isso este atributo deve estar em igualdade com o atributo do WSDL;
- *exclude* – Permite excluir a operação da lista de operações expostas pelo serviço.

A utilização da anotação pode ser observada na Figura 47.

```
@WebMethod(operationName="consultarCliente")
ClienteVO consultarCliente(String cpf);
```

Figura 47 - Anotação @WebMethod

WebParam

A anotação *@WebParam* (Figura 48) é usada para definir um parâmetro de uma operação. Cada parâmetro é referente a um *part* da mensagem definida utilizada no parâmetro *input* da operação. O parâmetro name deve referenciar o nome do *part* da mensagem.

```
@WebMethod(operationName="consultarCliente")
ClienteVO consultarCliente(@WebParam(name="cpf")
                           String cpf);
```

Figura 48 - Anotação @WebParam

WebResult

A anotação *@WebResult* (Figura 49) é usada para definir o retorno de uma operação. O retorno é referente ao *part* da mensagem definida e utilizada no parâmetro *output* da operação.

```
@WebMethod(operationName="consultarCliente")
@WebResult(name="clienteView")
ClienteVO consultarCliente(@WebParam(name="cpf")
                           String cpf);
```

Figura 49 - Anotação @WebResult

Com a utilização dessas anotações em conjunto, é possível programar toda a interface do contrato elaborada. A Figura 50 e a Figura 51 apresentam a interface e a implementação do serviço TratarCliente, respectivamente.

```
package br.uniriotec.monpgrafia.servicos;

import javax.jws.WebMethod;

@WebService
@SOAPBinding(style = Style.RPC)
public interface TratarCliente {

    @WebMethod(operationName="incluirCliente")
    @WebResult(name="sucesso")
    boolean incluirCliente(@WebParam(name="clienteView")
            ClienteVO clienteVO);

    @WebMethod(operationName="consultarCliente")
    @WebResult(name="clienteView")
    ClienteVO consultarCliente(@WebParam(name="cpf")
            String cpf);

    @WebMethod(operationName="excluirCliente")
    @WebResult(name="sucesso")
    boolean excluirCliente(@WebParam(name="cpf")
            String cpf);

    @WebMethod(operationName="alterarCliente")
    @WebResult(name="sucesso")
    boolean alterarCliente(@WebParam(name="clienteView")
            ClienteVO clienteVO);
}
```

Figura 50 - Interface do serviço TratarCliente

```java
package br.uniriotec.monografia.servicos;

import javax.jws.WebService;

@WebService(
        endpointInterface = "br.uniriotec.monografia.servicos.TratarCliente",
        name = "TratarClienteDataService",
        targetNamespace = "http://uniriotec.monografia.br/TratarClienteDataService/service",
        serviceName = "TratarClienteDataService",
        portName = "TratarClienteDataServicePort",
        wsdlLocation = "WEB-INF/wsdl/TratarClienteDataServiceConcreto.wsdl"
)
public class TratarClienteImpl implements TratarCliente{

    private IClienteDAO dao;

    public TratarClienteImpl() throws WebServiceException{
        String name = "java:global/PropostaClienteEJB/ClienteDAO";
        Context context;
        try {
            context = new InitialContext();

            dao = (IClienteDAO) context.lookup(name);
        } catch (NamingException e) {
            Log.getLog().fatal("Falha no construtor. Nao foi possivel injetar o DAO de Cliente", e);
            throw new WebServiceException("Falha no construtor. Nao foi possivel injetar o DAO de Cliente");
        }
    }

    @Override
    public boolean incluirCliente(ClienteView clienteView){
        try {
            Cliente cliente = transformaClienteViewEmCliente(clienteView);
            dao.gravar(cliente);
            return true;
        } catch (EntityExistsException e) {
            return false;
        } catch (IllegalArgumentException e) {
            return false;
        } catch (Exception e) {
            throw new WebServiceException(e.getMessage());
        }
    }
```

Figura 51 – Trecho da implementação do serviço TratarCliente

É importante observar o uso do localizador de serviços (método *lookup* na Figura 51) na implementação do serviço. Neste caso, o serviço é um objeto DAO para acesso a dados. Esta abordagem faz parte do padrão *Service Locator*, onde a classe desenvolvida, neste caso o serviço, não precisa conhecer as complexidades envolvidas para instanciação de outro objeto [Alur *et al.*, 2003]. A especificação JEE6 permite o uso deste tipo de injeção através da API JNDI (*Java Naming and Directory Interface*), utilizando propriedades de nomenclatura do servidor de aplicação para encontrar o EJB publicado informando: ip, porta e um nome único dado pelo servidor de aplicação durante o *deploy* de cada EJB. Neste trabalho, os EJBs foram instanciados através *lookup* nos construtores dos web services implementados, como pode ser observado na classe TratarClienteImpl.

4.2.5 Composição de serviços

A composição de serviços é a utilização de um ou mais serviços para gerar um novo serviço. Conforme abordado na seção 2.9, é possível compor serviços através de coreografia ou orquestração, sendo a última mais utilizada. O padrão para composição de serviços com orquestração para web services é o padrão WS-BPEL. Entretanto, o estudo dessa abordagem está fora do escopo deste trabalho. Neste trabalho, a composição de serviço foi feita utilizando a interface de serviços já publicados em classes que utilizam a anotação *@WebServiceClient*, fornecendo uma instância destes serviços. As interfaces dos serviços cliente especificam o *endpoint* de onde estão publicados. O serviço responsável pela composição utiliza o serviço cliente diretamente para invocar suas operações.

O serviço ManterCadastroCliente é um exemplo desse tipo de composição. A Figura 52 apresenta a classe TratarCliente, que representa a interface do serviço TratarClienteDataService que é utilizado na composição. A Figura 53 apresenta a classe responsável pela obtenção de uma instância desse serviço para ser utilizada no serviço ManterCadastroCliente. Como é possível observar na Figura 54, no momento que o serviço ManterCadastroClienteLogicSevice é iniciado, este obtém uma instância do serviço TratarClienteDataService.

```
package br.uniriotec.monografia.cliente;

import javax.jws.WebMethod;

@WebService(
        name = "TratarClienteDataService",
        targetNamespace = "http://uniriotec.monografia.br/TratarClienteDataService/defs")
@SOAPBinding(style = Style.RPC)
@XmlSeeAlso({
    br.uniriotec.monografia.vo.ObjectFactory.class
})
public interface TratarCliente {

    @WebMethod(operationName="incluirCliente")
    @WebResult(name="sucesso")
    boolean incluirCliente(@WebParam(name="clienteView")
            ClienteVO clienteVO);

    @WebMethod(operationName="consultarCliente")
    @WebResult(name="clienteView")
    ClienteVO consultarCliente(@WebParam(name="cpf")
            String cpf);

    @WebMethod(operationName="excluirCliente")
    @WebResult(name="sucesso")
    boolean excluirCliente(@WebParam(name="cpf")
            String cpf);

    @WebMethod(operationName="alterarCliente")
    @WebResult(name="sucesso")
    boolean alterarCliente(@WebParam(name="clienteView")
            ClienteVO clienteVO);
}
```

Figura 52 - Interface do serviço utilizado na composição do ManterCadastroCliente

```java
package br.uniriotec.monografia.cliente;

import java.net.MalformedURLException;


@WebServiceClient(
        name = "TratarClienteDataService",
        targetNamespace = "http://uniriotec.monografia.br/TratarClienteDataService/service",
        wsdlLocation = "http://localhost:8080/TratarCliente/TratarClienteDataService?wsdl"
)
public class TratarClienteService extends Service{

    private final static URL TRATARCLIENTEDATASERVICE_WSDL_LOCATION;
    private final static Logger logger =
        Logger.getLogger(br.uniriotec.monografia.cliente.TratarClienteService.class.getName());

    static {
        URL url = null;
        try {
            URL baseUrl;
            baseUrl = br.uniriotec.monografia.cliente.TratarClienteService.class.getResource(".");
            url = new URL(baseUrl, "http://localhost:8080/TratarCliente/TratarClienteDataService?wsdl");
        } catch (MalformedURLException e) {
            logger.warning("Falha ao criar URL para o WSDL localizado em: " +
                    "'http://localhost:8080/TratarCliente/TratarClienteDataService?wsdl'");
            logger.warning(e.getMessage());
        }
        TRATARCLIENTEDATASERVICE_WSDL_LOCATION = url;
    }
    protected TratarClienteService(URL wsdlDocumentLocation, QName serviceName) {
        super(wsdlDocumentLocation, serviceName);
    }

    public TratarClienteService() {
        super(TRATARCLIENTEDATASERVICE_WSDL_LOCATION, new QName(
                "http://uniriotec.monografia.br/TratarClienteDataService/service",
                "TratarClienteDataService"));
    }

    @WebEndpoint(name = "TratarClienteDataServicePort")
    public TratarCliente getTratarClienteDataServicePort() {
        return super.getPort(new QName("http://uniriotec.monografia.br/TratarClienteDataService/service",
                "TratarClienteDataServicePort"), TratarCliente.class);
    }
}
```

Figura 53 - Classe responsável pela obtenção do serviço TratarCliente

```java
package br.uniriotec.monografia.servicos;

import javax.jws.WebService;
import javax.xml.ws.WebServiceException;

import br.uniriotec.monografia.cliente.TratarCliente;
import br.uniriotec.monografia.cliente.TratarClienteService;
import br.uniriotec.monografia.vo.ClienteVO;

@WebService(
        endpointInterface = "br.uniriotec.monografia.servicos.ManterCadastroCliente",
        name = "ManterCadastroClienteLogicService",
        targetNamespace = "http://monografia.uniriotec.br/ManterCadastroClienteLogicService/service",
        serviceName = "ManterCadastroClienteLogicService",
        portName = "ManterCadastroClienteLogicServicePort",
        wsdlLocation = "WEB-INF/wsdl/ManterCadastroClienteLogicServiceConcreto.wsdl"
)
public class ManterCadastroClienteImpl implements ManterCadastroCliente {
    private TratarCliente tratarCliente;

    public ManterCadastroClienteImpl() {
        TratarClienteService service = new TratarClienteService();
        tratarCliente = service.getTratarClienteDataServicePort();
    }
```

Figura 54 – Composição de serviços

4.2.6 Testes de desenvolvimento

Após o desenvolvimento dos serviços, devem ser realizados os testes dos mesmos, antes que o projeto seja entregue a uma equipe de testes. Neste trabalho, os testes foram divididos em duas fases: testes unitários e testes funcionais.

4.2.6.1 Testes Unitários

Antes da implementação dos serviços as funcionalidades básicas do DAO genérico e as consultas avançadas passaram por testes unitários, garantindo que a camada de acesso a dados estivesse operacional. Para realizar esses testes foi utilizado o framework TestNG, que trabalha com uso de anotações para definir os testes.

Neste trabalho foram abordadas apenas as seguintes anotações da ferramenta:

- ***Test* –** Define a classe ou a função como um teste. É possível definir operações dependentes, onde uma determinada operação depende da execução de outra operação. Entretanto, esse recurso deve ser utilizado com cautela. Como exemplo, se a lógica do teste for construída considerando que uma operação sempre será executada com sucesso para em seguida testar outra operação, caso a primeira operação falhe por um motivo não esperado, pode ser complicado identificar onde se encontra o erro real, uma vez que o objetivo do teste é testar a segunda operação e não a primeria. Todas as classes e operações de teste foram anotadas *@Test*.
- ***BeforeClass* –** Define que a função será a primeira função executada no teste. Esta anotação permite que o EJB seja carregado através de lookup antes que as operações do DAO sejam executadas pelos testes. Isto permitiu também garantir que estas chamadas estavam de acordo com a JNDI e poderiam ser utilizadas posteriormente pelos serviços.

A Figura 55 apresenta um exemplo da classe de teste para o DAO de Proposta de Contrato.

```java
public class DAOPropostaContratoTest extends TestBase{

    IPropostaContratoDAO propostaContratoDAO;

    private long numeroContrato = 1234567;

    @BeforeClass
    protected void setUp() throws Exception {
        propostaContratoDAO = (IPropostaContratoDAO) context.lookup("java:global/ConcessaoCreditoEJB/PropostaContratoDAO");
    }

    @Test
    public void testInserirPropostaContrato(){
        PropostaContrato proposta = new PropostaContrato();
        proposta.setNumeroContrato(numeroContrato);
        proposta.setSituacao("aprovado");

        CreditoConcedido credito = new CreditoConcedido();
        credito.setId(new Long(123));
        credito.setNumeroParcelas(3);
        credito.setSituacao("aprovado");
        credito.setValorTotalConcedido(12352);

        proposta.setCreditoConcedido(credito);

        try {
            propostaContratoDAO.gravar(proposta);
        } catch (Exception e) {
            System.out.println(e);
            Assert.fail();
        }
    }

    @Test(dependsOnMethods={"testInserirPropostaContrato"})
    public void testCarregarPropostaContrato(){
        try {
            PropostaContrato proposta = propostaContratoDAO.carregar(numeroContrato);
            Assert.assertEquals("aprovado", proposta.getSituacao());
        } catch (Exception e) {
            Assert.fail();
        }
    }

    public void testAtualizarPropostaContrato(){

    public void testListarPropostaContratoPorSituacao(){

    public void testConsultarPropostaContratoPorIdCreditosConcedidos(){

    public void testExcluirCliente(){

}
```

Figura 55 - Classe de testes do DAO PropostaContrato

Após a execução do teste, a ferramenta apresenta um resumo das operações em que houve sucesso ou fracasso, conforme apresentado pela Figura 56.

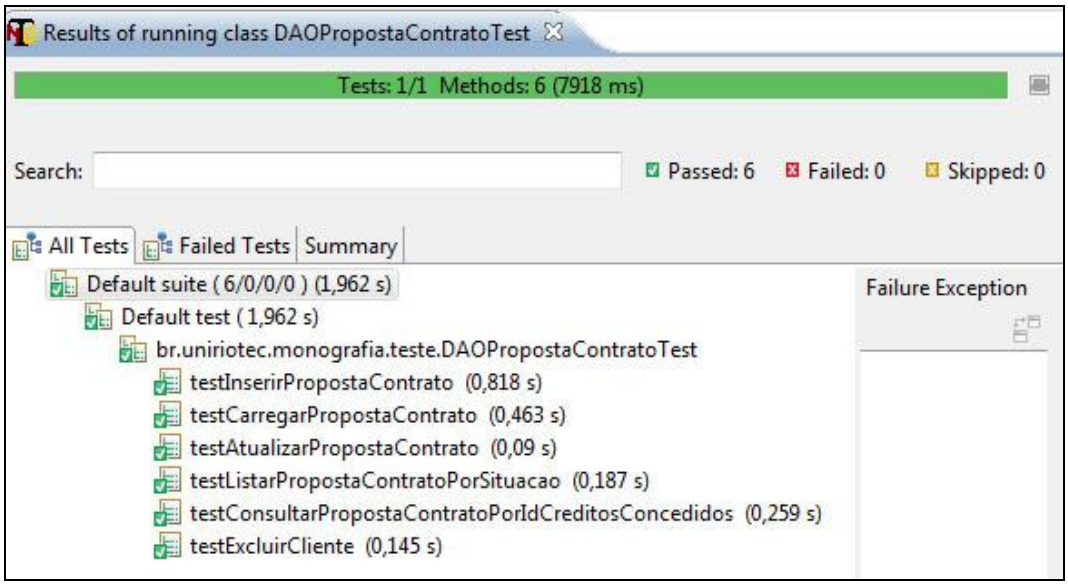

Figura 56 - Resultado da realização de um teste com TestNG

4.2.6.2 Testes Funcionais

Após a implementação de cada serviço, o mesmo passou por testes manuais, onde foram verificadas as invocações de cada operação do contrato e avaliado o retorno esperado das mesmas.

Para realização dos testes dos serviços, foi utilizada a ferramenta SoapUI, voltada principalmente para testes funcionais de web services. Para serviços já publicados em um servidor, basta criar um projeto no SoapUI com a URL do WSDL, e automaticamente serão criados *requests* para cada operação do serviço com as mensagens definidas. A partir daí, basta informar os valores dos parâmetros de acordo com o desejado, invocar uma chamada e será apresentada a resposta do serviço, seja de erro ou sucesso.

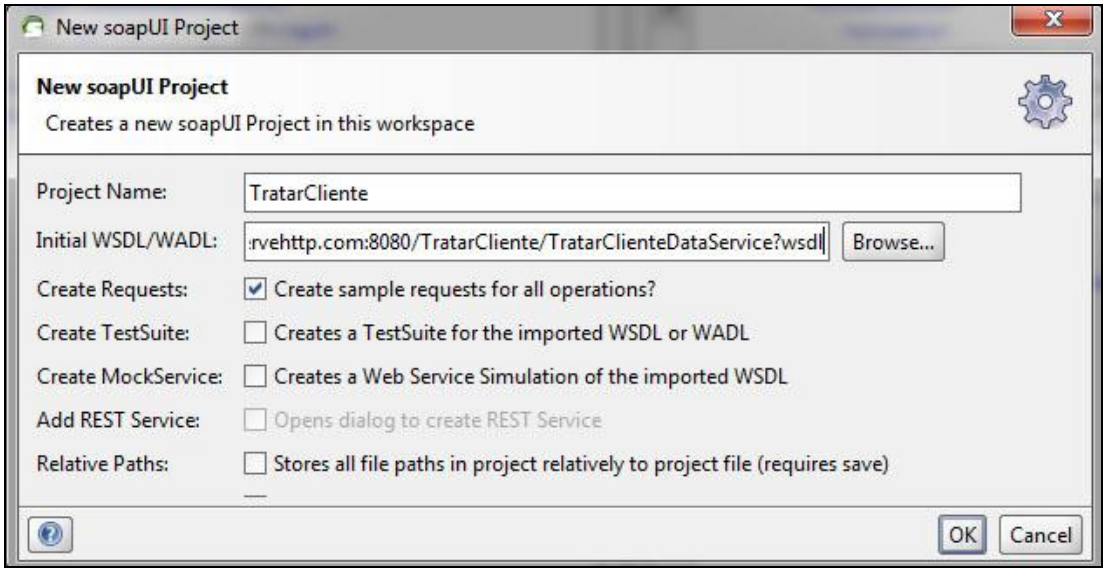

Figura 57 - Criação de Teste com SoapUI

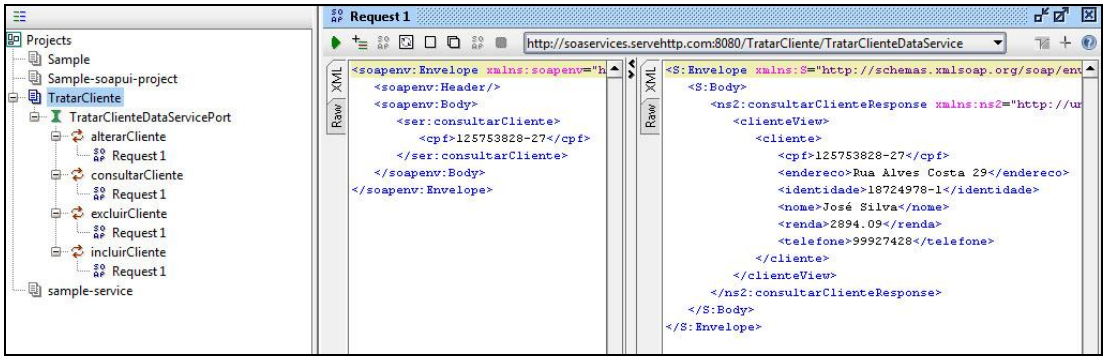

Figura 58 - Teste de chamada com SoapUI

A Figura 57 apresenta a criação de novos testes com a ferramenta, a partir de um wsdl publicado. É possível ainda gerar simuladores (Mocks) para o teste, caso o serviço não esteja em produção, mas ainda podendo simular resultados esperados. A Figura 58 mostra uma chamada da operação consultarCliente do serviço, onde a partir da requisição gerada automaticamente, é possível verificar a resposta do serviço para validar com o contrato.

4.3 Teste de serviço

Esta é a última etapa da fase de projeto do ciclo de vida do serviço. Neste momento, uma equipe de testes assume o papel e deve, não somente testar de novo e garantir aderência aos requisitos, como assegurar qualidade do serviço.

A ferramenta SoapUI também pode ser utilizada para realizar simulação e testes unitários de chamadas SOAP com objetivo de mitigar erros, falhas e assegurar o atendimento aos requisitos. Salvo os requisitos, os testes devem atender a métricas tais como acessos simultâneos, tempo de resposta e disponibilidade. Para isso, o SoapUI oferece também recursos específicos para garantia de qualidade, entre eles: casos de teste, testes de integração contínua, testes de carga e testes de segurança.

As análises e métricas coletadas pela equipe de testes devem servir de base para adquirir vantagens competitivas no mercado ou estipular garantias de contrato pelo provedor do serviço.

Finalizada esta fase, e o serviço tendo passado nas métricas definidas, a fase de projeto do ciclo de vida se encerra, e o mesmo torna-se habilitado para publicação e provimento.

4.4 Publicação de serviço

4.4.1 Compilação de serviço com Ant

Antes de publicar um web service, é preciso compilar e empacotar o projeto do serviço, incluindo ou referenciando as bibliotecas e/ou projetos associados para que possam ser publicados em um servidor de aplicação.

Em projetos simples, a construção manual é completamente viável. Porém, quando os projetos se tornam complexos, interdependentes e compartilhando bibliotecas em um servidor de aplicação, o trabalho manual pode se tornar lento e passível de erros. Neste caso, ferramentas devem ser utilizadas para automatizar a publicação dos serviços como, por exemplo, o Ant. O framework Ant é um automatizador simples, escalável e extensível para construção de projetos [Hatcher e Loughran, 2003]. Descrita em linguagem XML, a ferramenta possibilita facilidades como compilar código, copiar arquivos, referenciar bibliotecas, empacotar um projeto e publicar em um servidor de aplicação. A Figura 59 exemplifica o arquivo de compilação do serviço *Tratar Cliente*.

A Tabela 12 apresenta os principais elementos de um arquivo Ant. Já a Figura 59 mostra um exemplo prático de utilização dos recursos deste framework como compilação de classes, empacotamento e implantação automática no servidor.

Tabela 12 - Principais elementos Ant

Elemento	Descrição
path	Define o conjunto de bibliotecas (arquivos ou diretórios) necessários para compilação das classes
target	Define um conjunto de operações. Funcionamento similar à uma função.
war	Empacota um projeto na estrutura WAR (Web Application Resource)
jar	Empacota um projeto na estrutura JAR (Java Application Resource) para ser utilizado como biblioteca ou EJB.
javac	Compila uma ou um conjunto de classes.
exec	Permite executar linhas de comando. Funciona com comandos bat e shell.
antcall	Realiza uma chamada para outro *target* definido anteriormente.

```
<project name="TratarCliente" default="war" basedir=".">
    <description>
        Arquivo de construção do projeto do serviço TratarCliente.
    </description>

    <!-- Acesso ao arquivo de propriedades do build.-->
    <property file="build.properties" />

    <path id="classpath">
        <fileset dir="C:/glassfish3/glassfish/domains/domain1/lib/ext">
            <include name="*.jar" />
        </fileset>
        <fileset dir="C:/glassfish3/glassfish/lib">
            <include name="javaee-api-6.0.jar" />
        </fileset>
    </path>

    <target name="init" description="inicializa o ant">
        <deltree dir="${build}" />
        <mkdir dir="${build}" />
    </target>

    <target name="compile" depends="init" description="Compila as classes">
        <!-- Compila o código java -->
        <javac debug="true" destdir="${build}" source="1.6" target="1.6">
            <classpath refid="classpath" />
            <src path="${src}" />
        </javac>
    </target>

    <target name="war" depends="compile" description="Criar o war do projeto">

        <!-- Cria o diretório de distribuição do war. -->
        <delete dir="${dist}" failonerror="false" />

        <mkdir dir="${dist}/war" />

        <delete file="" />
        <!-- Copia a estrutura responsável pela identificação do serviço -->
        <copydir dest="${dist}/war/build/WEB-INF/" src="${webcontent}/WEB-INF/" />

        <war destfile="${dist}/TratarCliente.war" basedir="${dist}/war/build/">
            <metainf dir="${webcontent}/META-INF"/>
            <lib dir="${libs}" />
            <classes dir="${build}" />
        </war>
    </target>

    <target name="deploy" depends="war" description="Realiza a publicação(deploy) do serviço.">
        <echo>Instalando...</echo>
        <exec executable="cmd">
            <arg value="/c" />
            <arg value="asadmin" />
            <arg value="deploy" />
            <arg value="--user" />
            <arg value="admin" />
            <arg value="--passwordfile" />
            <arg value="glassfish_password.txt" />
            <arg value="${dist}/TratarCliente.war" />
        </exec>
    </target>
</project>
```

Figura 59 - Build.xml - Tratar Cliente

Ant é um framework extenso e permite uma série de outros artifícios de automação como integração com sistema de controle de versão (por exemplo, CVS), execução automática de testes unitários com JUnit , envio de e-mail, entre outros que não serão abordados neste trabalho.

4.4.2 Implantação do Serviço

Sendo o serviço um projeto web, o mesmo deve ser implantado em um servidor de aplicação web, para que então possa estar disponível para utilização e consumo.

Para isso, foi o escolhido o servidor GlassFish em sua versão 3.1.2, um servidor robusto, presente na especificação JEE 6, *open-source* e largamente utilizado no mercado. Além disso, possui uma interface completa de administração pelo navegador, que permite facilidade e praticidade para efetuar operações fundamentais, tais como:

- Criação de pool de EJB;
- Criação, configuração e teste de conexão com banco de dados;
- Criação de pool de conexões;
- Implantação de EJB;
- Implantação de Web Service.

4.5 Considerações sobre o desenvolvimento dos serviços

Este capítulo apresentou o desenvolvimento dos serviços e dos demais projetos que constituem a arquitetura do projeto. O diagrama de arquitetura que representa a relação de cada projeto e suas dependências é apresentado na Figura 60.

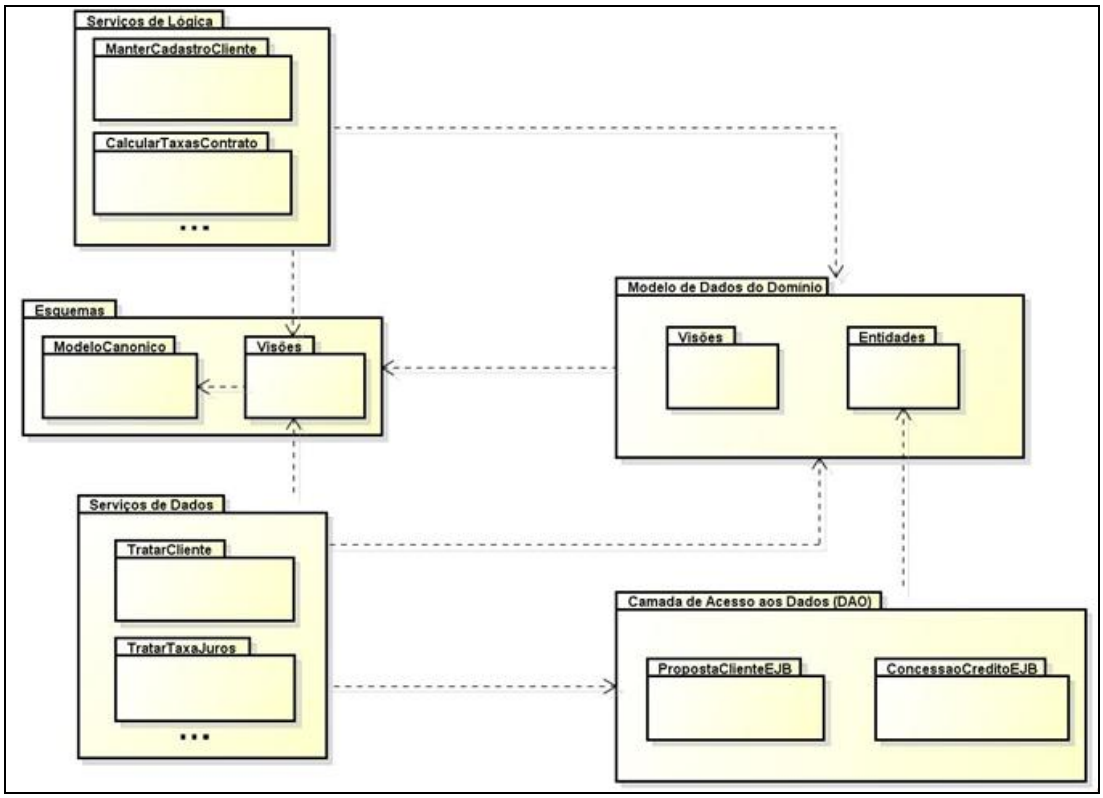

Figura 60 - Diagrama de Arquitetura do Projeto

O modelo canônico elaborado do processo de análise de crédito foi matéria-prima para a elaboração de um esquema centralizado de tipos de dados e os esquemas de visões propostos para cada entidade presente no modelo canônico.

Os WSDLs foram criados manualmente considerando: as exigências do BP; o padrão sugerido por Azevedo *et al*. [2009a], que define como serviços de dados aqueles que executam apenas operações de persistência e serviço de lógica; aqueles responsáveis pelo processamento de regras de negócio da organização, mas que possam porventura fazer acesso a dados para seu processamento; o padrão de projeto de modularização, que proporciona um desacoplamento da parte conceitual em relação à parte física do WSDL; e os tipos de dados dos esquemas de visão definidos para o projeto.

Após o desenvolvimento do contrato dos serviços, foi necessário não somente o desenvolvimento dos web services, mas a elaboração de uma arquitetura que desse suporte ao desenvolvimento dos mesmos. Inicialmente foi escolhido o servidor de aplicação GlassFish devido a sua robustez e facilidade de configuração de pool de conexões. Em seguida, fez-se necessário o mapeamento do esquema centralizado de

dados e suas visões para classes Java. Assim, foi criado o projeto ModeloCanonico contendo as classes Java (POJO) que representam simultaneamente as entidades que devem ser persistidas em banco, os tipos de dados do esquema centralizado utilizados na transferência de dados entre cliente e servidor e as classes (POJO) que representam cada um dos esquemas de visão elaborados.

Após a definição e criação das classes que representam o modelo canônico, foram desenvolvidos dois projetos EJB responsáveis pela persistência dos dados: PropostaClienteEJB, responsável pelas operações de persistência das entidades Cliente, Funcionario, Peca e PropostaCredito; e ConcessaoCreditoEJB, responsável pelas operações de persistência das entidades CreditoConcedido, Parcela, PropostaContrato, TaxaContrato e TaxaJuros. A escolha de utilização de EJBs se deu pela característica destes serem escaláveis, transacionais, terem facilidade de distribuição e permitir injeção automática de contexto de persistência (EntityManager), eliminando necessidade de criação de fábrica de conexões. Foi desenvolvida uma classe utilizando o padrão DAO para cada entidade tratada pelos EJBs, o que garantiu a separação da camada de persistência de dados da implementação dos serviços. Tanto o projeto PropostaClienteEJB quanto o projeto ConcessaoCreditoEJB têm uma dependência com o projeto ModeloCanonico, onde foram desenvolvidas as entidades que devem sofrer as operações de persistência.

Os web services foram implementados utilizando o framework JAX-WS devido à facilidade de desenvolvimento utilizando anotações, integração direta com o framework JAX-B e por sua aderência ao BP. Utilizando as anotações presentes no JAX-WS foi possível fazer o mapeamento do WSDL gerado manualmente para as classes Java (POJO) desenvolvidas. Devido à centralização do desenvolvimento das classes que representam as visões e entidades, cada projeto de serviço desenvolvido tem uma dependência com o projeto ModeloCanonico. Projetos de serviços que precisam efetuar operações de persistência de uma determinada entidade através do seu respectivo DAO utilizaram o padrão *Service Locator*. A abordagem utilizada foi a instanciação através de *lookup*, onde uma instância do DAO é obtida através do localizador de serviços JNDI. Desta forma, o serviço não precisa ter conhecimento sobre a complexidade de criação do DAO e se a implementação do mesmo for alterada o serviço que o utiliza não será afetado.

É importante ressaltar que a dependência gerada com a utilização do padrão de centralização de esquema foi propagada para o desenvolvimento do esquema centralizado e suas visões na linguagem de programação Java (projeto ModeloCanonico). Contudo, os benefícios desse padrão também se fizeram presentes na codificação, proporcionando uma governança mais eficiente sobre as classes de entidades que devem ser utilizadas no processo de análise de crédito. Com o intuito de facilitar o desenvolvimento e implantação dos serviços, as classes presentes no projeto ModeloCanonico foram importadas em um arquivo .jar para a pasta [GLASSFISH_HOME]/domains/analise_credito/lib/ext. Desta forma, todos os projetos presentes no domínio analise_credito podem utilizar as classes do ModeloCanonico sem tê-las diretamente em seus projetos.

Finalmente, após o desenvolvimento dos EJBs (que contém os DAOs das entidades) e dos serviços, estes foram compilados com suas devidas dependências e empacotados utilizando a ferramenta de automatização de publicação Ant. Após publicação no servidor de aplicação GlassFish, foram testados. Os EJBs passaram por testes unitários utilizando o framework TestNG; já os web services foram testados funcionalmente utilizando a ferramenta SoapUI.

Capítulo 5: Conclusão

Web services são desenvolvidos majoritariamente utilizando a abordagem code-first. Apesar de facilitar e agilizar o desenvolvimento dos serviços, esta técnica traz impactos negativos à governança de uma arquitetura orientada a serviços, tais como:

- Maior acoplamento dos serviços com as tecnologias que os implementaram;
- Baixo índice de reusabilidade dos serviços;
- Maior impacto nos consumidores após mudanças;
- Baixo grau de durabilidade devido à necessidade de geração de novo contrato a cada manutenção;
- Impossibilidade de utilização de versionamento;
- Impossibilidade de utilização de tipos de dados centralizados.

Neste trabalho, foi abordada a utilização do padrão de desenvolvimento contract-first como alternativa ao code-first e suas implicações negativas. Para isso, foi dada seqüência ao desenvolvimento dos serviços candidatos identificados e projetados em Souza *et al* [2011], dando seguimento ao ciclo de vida de desenvolvimento de serviços proposto por Gu e Lago [2007].

Em uma abordagem contract-first, o contrato do serviço é desenvolvido orientado ao negócio, para que depois seja codificado. Assim, a confecção do contrato antes da codificação possibilitou um acoplamento positivo de lógica para contrato em detrimento do acoplamento negativo do contrato com uma tecnologia específica [Erl, 2008]. Utilizando code-first, também é possível desenvolver orientado ao negócio. Entretanto, neste caso, haverá um acoplamento com a tecnologia, pois o contrato será gerado a partir da codificação, causando uma dependência em relação à linguagem ou *framework*.

Devido ao baixo acoplamento possibilitado pelo contract-first, é possível obter alto índice de durabilidade dos serviços desenvolvidos, uma vez que alterações na lógica do serviço podem ser realizadas sem que haja alteração do contrato. Em

code-first também é possível fazer alterações que não levem a mudanças no contrato, desde que se utilize na manutenção a mesma tecnologia utilizada no desenvolvimento inicial. Se o serviço foi desenvolvido utilizando linguagem Java e *framework* JAX-WS, sua manutenção deve ser feita utilizando a linguagem Java e o *framework* JAX-WS.

Neste trabalho, para que fosse possível promover interoperabilidade dos serviços, os mesmos foram desenvolvidos tomando como base os padrões de desenvolvimento de serviço presentes no *Basic Profile* (BP), que é o principal produto do consórcio *Web Service Interoperability* (WS-I). A utilização do BP garante homogeneidade diante das muitas possibilidades de configuração das especificações necessárias ao desenvolvimento de serviços. Essa padronização possibilita que um maior número de plataformas tecnológicas utilize os serviços.

O *framework* JAX-WS, utilizado para o desenvolvimento dos serviços, está em conformidade com o BP. Isso significa que tanto utilizando contract-first quanto utilizando code-first é possível obter interoperabilidade. Entretanto, conforme demonstrado, a utilização de contract-first permitiu não apenas a conformidade com o BP como permitiu a utilização dos tipos de dados em esquemas centralizados e a modularização do WSDL.

A padronização de tipos de dados por domínio (centralização de esquema), desenvolvido a partir do modelo canônico proposto em Souza *et al* [2011], proporcionou a eliminação de tipos de dados redundantes, melhorando a governança sobre os tipos de dados, tornando desnecessária transformações de dados e, conseqüentemente, melhorando o desempenho e interoperabilidade dos serviços.

O desempenho também foi otimizado devido à utilização do padrão Jardim do Éden, associado ao esquema de visões elaborado no projeto. Essas visões proporcionaram diminuição do tráfego de informações em mensagens SOAP, pois apenas os dados definidos no esquema de visão são trafegados. Entretanto, não foi possível comprovar a melhoria de desempenho relativa às validações diretamente no esquema sem a necessidade de criação de uma instância do serviço. As chamadas do serviço iniciavam uma instância do mesmo, sem que fosse bloqueado pela validação do esquema, fazendo com que as validações ocorressem apenas na codificação, assim como seria em code-first.

Já a modularização do WSDL possibilitou a separação da parte concreta da parte abstrata do contrato do serviço, o que é uma boa prática de projeto em desenvolvimento de software. Hewitt [2009] menciona que é possível utilizar diferentes estilos de *binding*, codificação e *endpoint* para uma mesma interface de serviço. Entretanto, utilizando o *framework* JAX-WS 2.2 não foi possível utilizar diferentes estilos de *binding*, uma vez que a definição do estilo deve ser configurada diretamente na interface do serviço. Apesar disso, o *framework* possibilita codificações e *endpoints* diferentes para uma mesma interface. Isto possibilita implementações de regras ou requisitos distintos, como por exemplo, serviços em diferentes idiomas, restrição de informações de acordo com o perfil, elaboração de conteúdo de acordo com a região de acesso, entre outros.

Durante o desenvolvimento dos serviços, foi possível identificar melhorias nas fases de análise e projeto. Seguem abaixo as oportunidades de melhorias observadas:

- A heurística de cluster do método de identificação dos serviços candidatos prevê apenas operações básicas de dados (CRUD) que devem ser implementadas em cada serviço identificado. Contudo, de acordo com o objetivo da modelagem de processos, o detalhamento da modelagem pode não conter todos os detalhes das operações necessárias às implementações dos serviços. Desta forma, a definição de operações de serviços na fase de projeto deve considerar a identificação de novas operações como serviços necessários aos serviços de dados.
- Foi observado que o serviço de cálculo de taxas de contrato foi identificado tanto pela heurística de múltiplas instâncias (Tabela 7, índice 37) quanto pela heurística de requisitos de negócio (Tabela 7, índice 34). Durante a implementação, um dos serviços foi escolhido e removido. Dessa forma, é necessário, durante a fase de projeto, que um projetista analise os serviços identificados a fim de remover duplicidades que não puderam ser identificadas durante a aplicação da heurística de eliminação de serviços duplicados.

Foi possível seguir a abordagem contract-first proposta com o uso do framework JAX-WS. No entanto, algumas dificuldades ou bugs foram identificados durante o processo do desenvolvimento:

- Apesar de o estilo *RPC* estar de acordo com o WS-I BP, o estilo mais utilizado é *Document*. Entretanto, os serviços desenvolvidos utilizando este estilo não funcionaram, pois durante a tentativa de consumo ocorreu erro de despacho e a operação não foi encontrada. Assim, foi decidido implementar os serviços utilizando o estilo *RPC*, que não apresentou o mesmo problema. Vale ressaltar que utilizando code-first, o estilo *Document* funciona perfeitamente;
- A utilização de listas com estilo RPC e abordagem contract-first retornava mensagens vazias ou parâmetros de entrada vazios. Desta forma, para superar este problema foi necessária a substituição de listas por arrays. Foi validado o que ocorreria se fosse utilizada abordagem code-first e foi verificado que é possível a utilização de listas perfeitamente neste caso;
- Foi detectado um problema na composição de serviços utilizando as anotações para consumo de serviço do JAX-WS. Ao iniciar o servidor de aplicação Glassfish, o mesmo entende que existe uma duplicata do serviço que está sendo registrado. No entanto, não foi possível precisar se este problema é relativo ao framework ou a alguma configuração no servidor de aplicação.

Como proposta para trabalhos futuros, o desenvolvimento dos serviços poderia ser melhorado com a utilização de BPEL para composição dos serviços, publicação dos serviços básicos e compostos em um ESB e registro dos mesmos em um UDDI.

Referências Bibliográficas

ALUR D., CRUPI, J., MALKS, D., 2003, **Core J2EE Patterns: Best Practices and Design Strategies**. Prentice Hall / Sun Microsystems Press.

AZEVEDO, L.G., BAIAO, F., SANTORO, F., SOUZA, J., REVOREDO, K., PEREIRA, V., HERLAIN, I. 2009a. **Identificação de serviços a partir da modelagem de processos de negócio**. In: Simpósio Brasileiro de Sistemas de Informação (SBSI), Brasília.

AZEVEDO, L.G., SANTORO, F., BAIAO, F., SOUZA, J., REVOREDO, K., PEREIRA, V., HERLAIN, I., 2009b. **A Method for Service Identification from Business Process Models in a SOA Approach**. In: 10th International Workshop on Business Process Modeling, Development, and Support (BPMDS), 2009, Amsterdam. Enterprise, Business-Process, and Information Systems Modelling. v. 29.

AZEVEDO, L.G., SOUZA, J.F., SANTORO, F ., BAIÃO, F. 2009c. **Metodologia para Análise e Projeto de Serviços em uma abordagem SOA.** Relatórios Técnicos do DIA/UNIRIO (RelaTe-DIA), 0023/2009.

AZEVEDO, L.G., SOUSA, H.P., SOUZA, J.F., SANTORO, F., BAIÃO, F. 2009d. **Identificação Automática de Serviços Candidatos a partir de Modelos de Processos de Negócio.** In: Conferencia IADIS Ibero Americana WWW/INTERNET 2009, Alcalá de Henares, Madri,Espanha.

AZEVEDO, L.G., SOUZA, J.F., SANTORO, F ., BAIÃO, F. 2009e. **Metodologia para Análise e Projeto de Serviços em uma abordagem SOA.** Relatórios Técnicos do DIA/UNIRIO (RelaTe-DIA), 0023/2009.

AZEVEDO, L.G., SOUZA, J.F., BAIÃO, F., SANTORO, F. 2009f. **Inspeção da Ferramenta Oracle BPEL PM.** Relatórios Técnicos do DIA/UNIRIO (RelaTe-DIA), 0016/2009.

AZEVEDO, L. G., BAIAO, F., SANTORO, F. SOUZA, J. F., (2011) **"A Business Aware Service Identification and Analysis Approach".** In: IADIS International Conference Information Systems 2011, March, 11-13, Avila, Spain.

BURKEE, B. MONSON-HAEFEL, R., 2007. **Enterprise JavaBeans 3.0**. O´Reilly.

CHINNICI, R., MOREAU, J.-J., RYMAN, A., WEERAWARANA, S., 2007. **Web Services Description Language (WSDL) Version 2.0.** World Wide Web Consortium (W3C). Disponível em <http://www.w3.org/TR/wsdl20/>. Acessado em Junho de 2012.

CHRISTENSEN, E., CURBERA, F., MERDITH, G., WEERAWARANA, S. 2001. **Web Services Description Language (WSDL) 1.1**. World Wide Web Consortium (W3C). Disponível em <http://www.w3.org/TR/wsdl>. Acessado em Junho de 2012.

DAMASCENO, J. C., **Introdução à Composição de Serviços Web**. Disponível em: http://www.ufpi.br/subsiteFiles/ercemapi/arquivos/files/minicurso/mc8.pdf. Acesso em 05 jun. 2012.

DIIRR, T.; SOUZA, A.; AZEVEDO, L.; SANTORO, F. **Modelo de Processos de Negócio Analisar Pedido Crédito**. Relatórios Técnicos do DIA/UNIRIO (RelaTe-DIA), RT-0008/2010, 2010. Disponível (também) em: http://seer.unirio.br/index.php/monografiasppgi.

ERL, T., 2004, **Oriented Architecture: A Field Guide to Integrating XML and Web Services.** Prentice Hall.

ERL, T., 2005, **Service-Oriented Architecture: concepts, technology, and Design**, Prentice Hall.

ERL, T., 2008, **SOA Principles of Service Design**, Prentice Hall.

FATINATO, M. **Uma abordagem baseada em características para o estabelecimento de contratos eletrônicos para services web.** Tese de Doutorado, Unicamp, 2007.

GONCALVES, A., 2009, **Beginning Java™ EE 6 Platform with GlassFish™ 3**, Apress.

GU, Q., LAGO, P**., A stakeholder-driven service life cycle model for SOA, 2007.**

HANSEN , M. D., 2007, **SOA Using Java Web Services.**

HATCHER, E., LOUGHRAN, S., 2003, **Java Development with Ant,** Manning.

HEWITT, E., **Java SOA Cookbook**, 2009. O´Reilly.

IYENGAR, ASHOK., JESSANI, VINOD., CHILANTI, MICHELE, **WebSphere Business Integration Primer, Process Server, BPEL, SCA and SOA,** 2008.

JOSUTTIS, N. M., **SOA in Practice – The Art of Distributed System Design, Beijing**; Cambridge; Farnham; Köln; Paris; Sebastopol; Taipei; Tokyo: O'Reilly, 2007, 324 p.

JURIC, M., **A Hands-on Introduction to BPEL**, 2007. Disponível em: http://www.oracle.com/technetwork/articles/matjaz-bpel1-090575.html. Acesso em 05 jun. 2012.

KALIN, M., **Java Web Services: Up and Running**, 2009. O´Reilly.

KELLER, G, TEUFEL T., **SAP R/3 Process Oriented Implementation**, Addison-Wesley, 1998.

MAGALHAES, A. et al.: **Uma Estratégia para Gestão Integrada de Processos e Tecnologia da Informação através da Modelagem de Processos de Negócio em Organizações.** Revista Cientefico – Faculdade Ruy Barbosa. 45-53 (2007)

MARKS, E., BELL, M. **Service-Oriented Architecture: A Planning and Implementation Guide for Business and Technology**, 2006, Wiley.

METHA, E. **Java Architecture for XML Binding (JAXB),** 2003. Disponível em: http://www.oracle.com/technetwork/articles/javase/index-140168.html. Acesso em 09 jun. 2012.

MELLQVIST, P., **Don't repeat the DAO!**, 2006. Disponível em: http://www.ibm.com/developerworks/java/library/j-genericdao/index.html. Acesso em 12 jun. 2012.

MINISTRO, A., LEÃO, F., FARIA, F., LOPES, S., AZEVEDO, L.G., 2011. **BPEL: Principais Conceitos e Uso Prático,** Relatórios Técnicos do DIA/UNIRIO (RelaTe-DIA), 0011/2011.

OASIS, **Modelo de Referência para Arquitetura Orientada a Serviço 1.0,** disponível em http://www.pcs.usp.br/~pcs5002/oasis/soa-rm-csbr.pdf, OASIS, 2006.

PELTZ, C., Web Services Orchestration and Choreography, 2003. Disponível em <http://soa.sys-con.com/node/39800>. Acesso em 25 ago. 2012.

PUNTAR, S., LENDRIKE, A., MAGDALENO, A., BAIÃO, F., SANTORO, F., 2009. **Estudo Conceitual sobre BPMS.** Relatórios Técnicos do DIA/UNIRIO (RelaTe-DIA), 0007/2009.

QuickStartTutorial, Oracle BPEL Process Manager 10.1.2.0.x Quick Start Tutorial, 2009. Disponível em http://download.oracle.com/otndocs/products/bpel/quickstart.pdf. Acesso em 29 mai. 2009.

SCHEER, A.-W., 2000. **ARIS - Business Process Modelling**. Springer, Berlin, Alemanha.

SHARP, A., McDermott, P., 2001. **Workflow Modeling: Tools for Process Improvement and Application Development.** Artech House.

SOSNOSKI, D., **Serviços da Web Java : Entendendo e Modelando o WSDL 1.1**, 2011. Disponível em <http://www.ibm.com/developerworks/br/library/j-jws20/index.html>. Acessado em 4 Jun. 2012.

THOM, L., IOCHPE, C., REICHERT, M.: **Workflow Patterns for Business Process Modeling.** In: 8th Int. Workshop on Business Process Modeling, Development, and Support (BPMDS), pp. 349–358 (2007)

VAJJHALA, S., FIALLI, J., 2006, **The Java™ Architecture for XML Binding (JAXB) 2.0.**

VASILIEV, Y., **SOA and WS-BPEL: Composing Service-Oriented Solutions with PHP and ActiveBPEL**. Birmingham: Packt Publishing, 2007.

WYKE, R.A., WATT, A. **, 2002, XML Schema Essentials.**

Apêndice 1 – Padrões de Projeto para Centralização de Esquema

1. Boneca Russa

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://ns.soacookbook.com/russiandoll"
    xmlns:tns="http://ns.soacookbook.com/russiandoll"
    elementFormDefault="unqualified">
    <xsd:annotation>
        <xsd:documentation>
            Book schema as Russian Doll design.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:element name="book">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="title" type="xsd:string" />
                <xsd:element name="price" type="xsd:decimal" />
                <xsd:element name="category" type="xsd:NCName" />
                <xsd:choice>
                    <xsd:element name="author" type="xsd:string" />
                    <xsd:element name="authors">
                        <xsd:complexType>
                            <xsd:sequence>
                                <xsd:element name="author" type="xsd:string"
                                    maxOccurs="unbounded" />
                            </xsd:sequence>
                        </xsd:complexType>
                    </xsd:element>
                </xsd:choice>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

2. **Fatia de Salame**

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://ns.soacookbook.com/salami"
    xmlns:tns="http://ns.soacookbook.com/salami"
    elementFormDefault="qualified">
    <xsd:annotation>
        <xsd:documentation>
            Book schema as Salami Slice design.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:element name="book">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="tns:title" />
                <xsd:element ref="tns:author" />
                <xsd:element ref="tns:category" />
                <xsd:element ref="tns:price" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="title" />
    <xsd:element name="price" />
    <xsd:element name="category" />
    <xsd:element name="author" />
</xsd:schema>
```

3. Veneziana

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://ns.soacookbook.com/venetianblind"
    xmlns:tns="http://ns.soacookbook.com/venetianblind"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified">
    <xsd:annotation>
        <xsd:documentation>
            Book schema as Venetian Blind design.
</xsd:documentation>
    </xsd:annotation>
    <!-- Single global root element exposed -->
    <xsd:element name="book" type="tns:BookType" />
    <!-- The root is given a type that is defined here, using all externally
        defined elements. -->
    <xsd:complexType name="BookType">
        <xsd:sequence>
            <xsd:element name="title" type="tns:TitleType" />
            <xsd:element name="author" type="tns:AuthorType" />
            <xsd:element name="category" type="tns:CategoryType" />
            <xsd:element name="price" type="tns:PriceType" />
        </xsd:sequence>
    </xsd:complexType>
    <xsd:simpleType name="TitleType">
        <xsd:restriction base="xsd:string">
            <xsd:minLength value="1" />
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:simpleType name="AuthorType">
        <xsd:restriction base="xsd:string">
            <xsd:minLength value="1" />
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:simpleType name="CategoryType">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="LITERATURE" />
            <xsd:enumeration value="PHILOSOPHY" />
            <xsd:enumeration value="PROGRAMMING" />
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:simpleType name="PriceType">
        <xsd:restriction base="xsd:float" />
    </xsd:simpleType>
</xsd:schema>
```

4. Jardim do Éden

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://ns.soacookbook.com/eden" xmlns:tns="http://ns.soacookbook.com/eden"
    elementFormDefault="qualified">
    <xsd:annotation>
        <xsd:documentation>
            Book schema as Garden of Eden design.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:element name="book" type="tns:bookType" />
    <xsd:element name="title" type="xsd:string" />
    <xsd:element name="author" type="xsd:string" />
    <xsd:element name="category" type="xsd:string" />
    <xsd:element name="price" type="xsd:double" />
    <xsd:complexType name="bookType">
        <xsd:sequence>
            <xsd:element ref="tns:title" />
            <xsd:element ref="tns:author" />
            <xsd:element ref="tns:category" />
            <xsd:element ref="tns:price" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```