**UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO**
**CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA**

Relatórios Técnicos
do Departamento de Informática Aplicada
da UNIRIO
nº 0009/2010

# XML-Schema e Modelo de Dados em SOA

**Leonardo Guerreiro Azevedo**
**Flávia Santoro**
**Fernanda Baião**

Departamento de Informática Aplicada

UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
Av. Pasteur, 458, Urca - CEP 22290-240
RIO DE JANEIRO – BRASIL

# Projeto de Pesquisa

## Grupo de Pesquisa Participante


## Patrocínio


PETROBRAS

Relatórios Técnicos do DIA/UNIRIO, No. 0009/2010
Editor: Prof. Sean W. M. Siqueira Setembro, 2010

# XML-Schema e Modelo de Dados em SOA*

Leonardo Guerreiro Azevedo, Flávia Santoro, Fernanda Baião

Núcleo de Pesquisa e Prática em Tecnologia (NP2Tec)
Departamento de Informática Aplicada (DIA) – Universidade Federal do Estado do Rio de Janeiro (UNIRIO)

{azevedo, flavia.santoro, fernanda.baiao}@uniriotec.br

**Abstract.** XML (Extensible Markup Language) is the main language in a SOA environment. It is used in message exchanges, data type descriptions (XML-Schema), service description (WSDL), service orchestration (WS-BPEL) etc. XML is recognized as an independent platform for data type descriptions, and its use can guarantees SOA principles, such as cohesion, low coupling and reuse. This reports aims at presenting main concepts of XML-Schema and the related patterns for schema definition.

**Keywords**: Service-Oriented Architecture (SOA), XML-Schema, Canonical Model, SML-Schema patterns.

**Resumo**. XML (Extensible Markup Language) é a principal linguagem em uma SOA. Ela é utilizada para troca de mensagens, descrição de tipos de dados (XML-Schema), descrição de serviços (WSDL), orquestração de serviços (WS-BPEL) etc. XML é visto como uma forma independente de plataforma para descrição dos tipos de dados. No entanto, o bom uso deste poderoso artefato pode auxiliar a manter os princípios de uma arquitetura orientada a serviços, tais como coesão, baixo acomplamento e reuso. O objetivo deste relatório é apresentar os principais conceitos de XML-Schema e discutir os padrões de projeto para elaboração de esquemas.

**Palavras-**chave: Arquitetura Orientada a Serviços (SOA), XML-Schema, Modelo Canônico, Padrões para esquemas XML.

---

* Trabalho patrocinado pela Petrobras.

Sumário

Figuras

# 1 Introdução

XML (Extensible Markup Language (XML) [W3C, 2004, 2008; W3Schools, 2009] é o idioma de SOA (Service-Oriented Architecture). Ela é usada para *payloads* de mensagem, configuração de aplicações, configuração de instalação, descoberta, definição de políticas de *runtime*, e para representar linguagens de execução como, por exemplo, BPEL (Business Process Execution Language), como WS-BPEL [OASIS, 2007]. Interfaces de Web service são representadas usando XML na forma de WSDL (Web Service Description Language) [W3C, 2007], e XML é usado como mecanismo para transporte de dados em SOA [Hewitt, 2009].

Hewitt [2009] aponta que o uso de XML para projeto de modelo de dados provê uma fundamentação independente de implementação; liberta do uso de ontologias; e torna a modelagem menos sujeita às restrições de revendedores. O autor aponta ainda que com o uso de novos dispositivos de comunicação (telefones celulares, texto, mensagens instantâneas, email etc.), não só muda *como* nós nos comunicamos, mas também *o que* está sendo comunicado. Usar XML para troca de dados em SOA encoraja o uso de uma abordagem mais voltada para o modelo, oferecendo visões de dados contextuais que podem fluir, serem transformados e interagir através de variados serviços e camadas. Esta característica é muito importante, por exemplo, em ambientes heterogêneos, tal como em organizações onde existem silos de aplicações, cada uma atrelada a um modelo rígido de dados que deve ser acomodado a cada nova atualização ou manutenção.

Neste relatório, XML Schema [W3C, 2008] será estudado a partir da visão de SOA, a qual demanda código aberto e flexibilidade. XML Schemas são muito importantes para modelagem de dados, mas não são orientados a objetos. Eles têm o objetivo de capturar um modelo de dados ao invés de um modelo de objeto. Esquemas permitem projetistas criarem modelos muito parecidos com objetos e existe um grau de polimorfismo que pode ser alcançado via esquemas. Entretanto, o ponto é que serviços devem ser gerais o suficiente para que eles possam se comunicar através de diferentes linguagens. Para obter esta interoperabilidade é melhor ser conservativo [Hewitt, 2009].

Deixar a interface simples ajudará a garantir a minimizar problemas futuros. Entretanto, existem diferentes modos de projetar esquemas que podem ajudar ou dificultar o trabalho ao longo do tempo. Estes modos devem ser estudados com cuidado, o que se propõe este relatório.

O objetivo deste relatório é apresentar os conceitos de XML Schema e modelo de dados em uma arquitetura orientada a serviços, identificando propostas de uso do modelo canônico para desenvolvimento de serviços.

Este relatório foi produzido pelo Projeto de Pesquisa em SOA como parte das iniciativas dentro do contexto do Projeto de Pesquisa do Termo de Cooperação entre NP2Tec/UNIRIO e a Petrobras/TIC-E&P/GDIEP.

Esse relatório está organizado em 5 capítulos, sendo o capítulo 1 a presente introdução. No capítulo 2, são apresentadas definições e melhores práticas para projeto de esquema XML em SOA. No capítulo 3, é apresentada a definição do modelo canônico e são apresentadas diretrizes e boas práticas para criação e representação do mesmo. Nos capítulos 4 e 5, são apresentadas as conclusões do trabalho e as referências bibliográficas, respectivamente.

# 2 Projeto de esquema XML em SOA

A flexibilidade de XML Schema, de certo modo, pode levar a dificuldades para perceber como escrevê-los de forma consistente e clara, a qual dará a combinação perfeita entre expressividade, flexibilidade, e tipos suficientemente fortes. Isto é agravado em uma abordagem SOA no modo que código Java será gerado e XML usado para troca de dados.

Schemas definem blocos de construção para definir entidades: tipos simples (*simple types*), tipos complexos (*complex types*), elementos (*elements*) e atributos (*attributes*). Os esquemas devem ser projetados com cuidado a fim de se obter serviços pouco acoplados que podem ser compostos usando orquestrações ou intermediado por ESBs (Enterprise Service Bus) [Hewitt, 2009] [Josuttis, 2007]. Em outras palavras, esquemas mal projetados podem levar a um grau enorme de acoplamento entre os serviços, levando à necessidade de re-projetar conjuntos inteiros de composições de serviços.

Esta seção discute questões relacionadas a projeto de esquemas XML em uma abordagem orientada a serviços.

## 2.1 Quando usar *element* ou *type*?

Ao definir um esquema para um item de dados, duas opções surgem: declarar o item como elemento ou declarar o item como um tipo. A Figura 1.a apresenta um exemplo do item *"Warranty"* declarado como elemento e Figura 1.b apresenta o mesmo item declarado como tipo.

| | |
|---|---|
| `<xsd:element name="Warranty">`<br>`...`<br>`</xsd:element>` | `<xsd:complexType name="Warranty">`<br>`...`<br>`</xsd:complexType>` |
| (a) | (b) |

**Figura 1 – Exemplo de item declarado como (a) elemento e (b) tipo**

Em [XFront, 2009b] são apresentadas as seguintes melhores práticas para decidir a respeito do uso de elemento ou tipo:

- O uso de tipo (*type*) é mais amplo, pois pode-se sempre criar um elemento a partir do tipo e, com um tipo, outros elementos podem reutilizar este tipo. Além disso, elementos e tipos são definidos em diferentes espaços de símbolos. Logo, pode-se ter um elemento e um tipo com o mesmo nome, caso tenha-se decidido inicialmente criar o tipo e mais tarde decidir criar o elemento com o mesmon nome. Por exemplo, pode-se declarar o tipo *Warranty* como apresentado na Figura 1.b, e mais tarde criar o elemento *Warranty*. O elemento *Warranty* é declarado então com o nome *Warranty* e sendo do tipo *Warranty*, como, por exemplo, da forma apresentada na .

```
<xsd:element name="Warranty" type="Warranty"/>.
```

**Figura 2 – Elemento *Warranty* do tipo *Warranty***

- Se um item de dados nunca será utilizado como um elemento em uma instância de documento, então defina-o como um tipo. Ou seja, se nunca existirá a instância apresentada Figura 3, então defina *Warranty* como um tipo (Figura 1.b).

```
<Warranty>
...
</Warranty>
```

**Figura 3 – Instância do item de dados *Warranty***

- Se a estrutura será reutilizada por outros elementos, então defina o item como sendo um tipo (Figura 4).

```
<xsd:complexType name="Warranty">
                ...
</xsd:complexType>
       ...
<xsd:element name="PromissoryNote" type="Warranty"/>
<xsd:element name="AutoCertificate" type="Warranty"/>
```

**Figura 4 – Definição do tipo *Warranty* e dos elementos deste tipo**

- Se o item será utilizado como elemento em documentos de instâncias e ele pode ser nulo em alguns casos e não ser nulo em outro casos, então defina o elemento como sendo um tipo.

  Na Figura 5.a é apresentada a declaração do elemento *Warranty* como sendo um elemento. Este elemento pode ser reutilizado, como apresentado na Figura 5.b. No entanto, ele não pode ser reutilizado como atributo *nillable* como *true*, pois esta é uma construção ilegal em XML Schema. Este problema é resolvido declarando-se *Warranty* como um tipo de dados. A Figura 6 apresenta um exemplo de declaração de *Warranty* como tipo e definição de elemento que pode receber valor nulo e de elemento que recebe valor não nulo.

| (a) | (b) | (c) |
|---|---|---|
| `<xsd:element name="Warranty">`<br><br>`...`<br>`</xsd:element>` | `<xsd:element ref="Warranty"/>` | `<xsd:element ref="Warranty" nillable="true"/>` |

**Figura 5 – (a) Declaração do elemento *Warranty*, (b) reuso do elemento, (c) construção ilegal atribuindo nulo**

```
<xsd:complexType name="Warranty">
              ...
</xsd:complexType>
```

```
<xsd:element name="Warranty" nillable="true" type="Warranty"/>
       ...
<xsd:element name="Warranty" type="Warranty"/>
```

**Figura 6 – Declaração de *Warranty* como tipo e definição de elemento que pode receber valor nulo e de elemento que recebe valor não nulo**

- Se o item poderá ser criado como um elemento e será permitido que ele seja substituído por outro tipo em instâncias de documento utilizando grupo de substituições, então é melhor definir o item de dados como sendo um elemento. Como exemplo, considere o grupo de substituição definido na Figura 7 e os elementos da Figura 8 que podem utilizados de forma intercambiável.

```
<xsd:element name="Warranty">
            ...
</xsd:element>
<xsd:element name="Guarantee" substitutionGroup="Warranty"/>
<xsd:element name="Promise" substitutionGroup="Warranty"/>
```

**Figura 7 – Grupo de substituição para *Warranty***

```
<xsd:Warranty>
      ...
</xsd:Warranty>
      ...

<xsd:Guarantee>
      ...
</xsd:Guarantee>
      ...

<xsd:Promise>
          ...
</xsd:Promise>
```

**Figura 8 – Autores de documentos podem utilizar as instâncias *Warranty*, *Guarantee* e *Promise* em documentos de acordo com o grupo de substituição**

## 2.2 Padrões de projeto para XML Schema

Hewitt [2009] apresenta cinco padrões de projeto para modelagem de esquemas: boneca russa (*Russian Doll*); fatia de salame (*Salami Slice*); Veneziana (*Venetian Blind*), Jardim do Éden (*Garden of Eden*); e Camaleão (*Chameleon*). A principal diferença entre estes padrões é se os elementos e tipos são globalmente definidos. Um elemento ou tipo global é aquele que é filho do nó *schema*. Um elemento ou tipo local é aquele que está aninhado dentro de outro elemento ou tipo. Um elemento local não pode ser utilizado em qualquer lugar.

### 2.2.1 Russian Doll

Na vida real, uma boneca russa é uma casca de madeira que atua como um container para outras bonecas idênticas, cada qual é uma casca contendo a boneca menor do que ela. Abrindo o topo da boneca mais de fora, revela a próxima boneca.

Já falando em XML, o padrão boneca russa corresponde à criação de um único elemento raiz global que contém todos os seus tipos constituintes; todos os elementos abaixo deste elemento são locais. Todas as declarações de elementos são aninhadas dentro do elemento raiz, e podem, portanto, serem utilizadas apenas uma vez, dento deste contexto. O elemento raiz é o único que usa o *namespace* global.

O padrão boneca russa tem as seguintes características:

- Ele tem apenas um único elemento raiz global;
- Todos os tipos são locais, ou seja, aninhados dentro do elemento raiz;
- Ele suporta apenas esquemas projetados inteiramente em um único arquivo;
- Ele tem alta coesão com acoplamento mínimo;
- Dado que tipos não são expostos, o esquema é totalmente encapsulado;
- Ele é o padrão mais simples de ler e escrever.

É importante ressaltar que, o padrão boneca russa não deve ser utilizado se é necessário reutilizar tipos.

A Figura 9 apresenta um exemplo de uso do padrão boneca russa. No esquema do exemplo, é definido um único elemento global, *book*. Os tipos necessários para criar um elemento *book* estão todos aninhados dentro dele. Elementos e tipos não podem ser

referenciados. O elemento *authors* redefine seus elementos filhos *author*. Este é um problema potencial de manutenção.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://ns.soacookbook.com/russiandoll"
           xmlns:tns="http://ns.soacookbook.com/russiandoll"
           elementFormDefault="unqualifie">
  <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>
```

**Figura 9 – Exemplo de uso do padrão boneca russa**

O padrão boneca russa tem algumas vantages. Para esquemas simples, ele é fácil de ler e escrever porque não existe mistura, não há referências, não há necessidade de procurar por definições de tipos. Não existe nenhuma flexibilidade de construção nesta forma de projeto, o que o torna previsível. Esquemas são desacoplados de outros esquemas. Logo, mudanças de tipos não afetam outras esquemas. O maior problema está na não possibilidade de reuso de tipos e elementos e no fato de que os esquemas podem ficar muito grandes.

Este padrão é apropriado, talvez, para projetar dados legados de fontes que são estáticas.

### 2.2.2 Salami Slice

O padrão fatia de salame representa o oposto ao projeto usando boneca russa. Este padrão propõe que todos os elementos sejam globais e que os tipos sejam locais. Dessa forma, todos os elementos ficam no namespace global, tornando o esquema reutilizável por outros esquemas. Cada elemento age como uma fatia de definição única, que pode ser combinada com outras definições.

O padrão boneca russa é rígido e inflexível, possibilitando a definição de um único elemento que ele define. Já fatia de salame é inteiramente aberto, permitindo uma grande variedade de combinações.

A Figura 10 apresenta um exemplo de uso do padrão 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>
```

**Figura 10 – Exemplo de uso do padrão fatia de salame**

O padrão fatia de salame tem as seguintes características:

- Todos os elementos são globais;
- Todos os elementos são definidos dentro de um *namespace* global;
- Todos os tipos são locais;
- Declarações de elementos nunca são aninhadas;
- Declarações de elementos são reutilizáveis. Fatia de salame oferece a melhor possibilidade de reuso dentre todos os padrões de projeto.
- É difícil determinar o elemento raiz, dado que existem várias possibilidades em potencial;
- O esquema é verborrágico. Tudo é claramente arranjado.

Como os elementos são globais, o esquema é reutilizável. No entanto, a mudança em um elemento pode afetar os elementos que o compõem. Portanto, o padrão fatia de salame é considerado fortemente acoplado.

### 2.2.3 Venetian Blind

O padrão veneziana é uma extensão do padrão boneca russa. O padrão define um único elemento raiz global para instanciação, e compõe ele com tipos definidos externamente.

A Figura 11 apresenta um exemplo de uso do padrão veneziana. Este padrão tem as seguintes características:

- Ele tem um único elemento raiz;
- Ele mistura declarações globais com declarações locais. No padrão boneca russa, todos os tipos são locais e no padrão fatia de salame todos os tipos são globais.

- Ele tem alta coesão, mas também acoplamento alto. Dado que seus componentes são acoplados e não alto contidos, ele pode ocasionar acoplamento entre esquemas;
- Ele maximiza reuso. Todos os tipos e o elemento raiz podem ser recombinados.
- Ele permite que sejam utilizados múltiplos arquivos para definir o esquema;
- Ele é verborrágico. Dividindo cada tipo desta forma permite controle de seletividade e granularidade para cada aspecto ou elemento individual, mas leva a muita digitação.

```
<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>

    <!—- Elemento raiz global único -->
    <xsd:element name="book" type="tns:BookType" />

    <!-- O tipo da raiz é definido aqui,
         usando todos os elementos definidos externamente.-->
    <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>

    <!-- Cada tipo usado pela raiz global é definido abaixo,
         e são potencialmente disponíveis para reuso dependendo do
         valor do atributo 'elementFormDefault' do esquema
    => se o valor for 'qualified', então os elementos são expostos
    => se o valor for 'unqualified', então os elementos são escondidos
    -->
    <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>
```

**Figura 11 – Exemplo de uso do padrão veneziana**

No exemplo da Figura 11, existem cinco tipos reutilizáveis: BookType, TitleType, AuthorType, CategoryType, PriceType e o elemento livro.

Escolha o padrão veneziana quando for necessário maximizar reuso e flexibilidade, e ter vantagens da exposição de *namespace*.

### 2.2.4 Garden of Eden

O padrão jardim do Éden foi identificado pela Sun Microsystem [Hewitt, 2009], e corresponde a uma combinação dos padrões fatia de salame e veneziana. Para tornar um esquema de acordo com este padrão, defina todos os elementos e tipos no *namespace* global, e então referencie elementos quando necessário.

A Figura 12 apresenta um exemplo do uso do padrão Jardim do Éden. Este padrão tem as seguintes características:

- Com este padrão obtém-se o maior reuso dentre os padrões. Elementos e tipos podem ser reutilizados livremente.
- Não existe encapsulamento de elementos neste esquema.
- Existem vários elementos raiz em potencial, dado que eles são globais. Diferente do padrão boneca russa, onde é claro qual é o elemento raiz, no jardim do Éden, as intenções do autor do documento são mascaradas.
- Entretanto, têm-se a maior flexibilidade possível. O autor do documento pode criar elementos de acordo com as necessidades dos usuários para ilustrar como eles poderiam criar seus elementos.

```
<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>
```

**Figura 12 – Exemplo de uso do padrão jardim do Éden**

De acordo com este exemplo, instâncias de documentos podem referenciar e usar o tipo *Book* diretamente, ou eles podem usar uma combinação de elementos.

### 2.2.5 Projeto de namespace camaleão

O padrão cameleão refere-se a projetar os tipos comuns de dados em um esquema sem usar nenhum *namespace*, chamado de camaleão. Define-se então um esquema "máster" que tem um namespace que inclui (<*include*>) o primeiro esquema. Os tipos comuns do esquema camaleão assumem o *namespace* definido no máster. A Figura 13 apresenta um esquema camaleão (sem *namespace*) e a Figura 14 apresenta a inclusão do esquema camaleão pelo esquema máster. O esquema máster define o tipo *Invoice* que inclui o *CustomerType* definido no esquema camaleão.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified">
<!-- Defined without namespace -->
    <xsd:complexType name="CustomerType">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

**Figura 13 – Esquema de cliente usando padrão camaleão**

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="urn:Invoice"
            xmlns="urn:Invoice"
            elementFormDefault="qualified">
    <xsd:include schemaLocation="CustomerChameleon.xsd"/>
    <!-- Invoice has a Product and a Customer -->
    <xsd:element name="Invoice">
        <xsd:complexType>
            <xsd:sequence>
            <!-- Define product here -->
                <xsd:element name="product">
                  <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="name" type="xsd:string"/>
                        <xsd:element name="sku" type="xsd:string"/>
                    </xsd:sequence>
                  </xsd:complexType>
                </xsd:element>
                <!-- Pull from Chameleon -->
                <xsd:element name="customer" type="CustomerType"
                             minOccurs="1" maxOccurs="1" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

**Figura 14 – Esquema máster incluindo o esquema camaleão**

O uso do esquema camaleão parece resolver os problemas de dependência de outros padrões de projeto de esquema. Se nenhum *namespace* for declarado para tipos comuns de dados sempre que possível, clientes não são impactados (*quebram*) se os tipos forem substituídos mais tarde.

O padrão de projeto de esquema camaleão é bastante discutido na literatura. Ele depende de certas áreas de especificação de esquema cuja interpretação não está completamente aceita entre fornecedores. Como o principal propósito de SOA é alcançar serviços genéricos interoperáveis, deve-se ter cuidado ao utilizar tipos que são suportados apenas algumas vezes.

Camaleão irá, em geral, degradar o desempenho durante validação, mesmo no caso de fornecedores que suportam o padrão. Isto porque o atraso na resolução de *namespace* não permite que *parsers* façam *cache* do esquema baseado nos seus namespaces.

Camaleão também limita a possibilidade de utilizar restrições de identidade XPath. XPaths não usam o *namespace* default, logo seu uso deve ser prefixado, o que desabilita o uso de camaleão.

Camaleão aumenta a possibilidade de colisão de nomes de componentes porque tipos não estão atrelados a seus *namespaces*.

*Namespace* existem para agrupar logicamente um conjunto de tipos e defini-los como partes de uma idéia geral. Se os tipos não podem ser colocados em um *namespace*, então deve-se reconsiderar o projeto em geral. Se é possível colocar os tipos em um *namespace*, mas deseja-se não fazer isto para ganhar flexibilidade, esteja ciente de problemas de interoperabilidade e colisão mais cedo ou mais tarde.

### 2.2.6 Considerações sobre os padrões

Em [XFront, 2009a] são apresentadas as seguintes melhores práticas em relação aos padrões boneca russa, fatia de salame e veneziana:

- O padrão veneziana deve ser escolhido quando o esquema requer flexibilidade na exposição de *namespace* com uma troca simples (*elementFormDefault="unqualified"*, para não explicitar, ou *elementFormDefault="qualified"*, para explicitar), e quando reuso é importante.
- Quando a tarefa requer tornar possível o uso de substituição de elementos, então é indicado o uso de fatia de salame.
- Quando o objetivo é minimizar acoplamento entre componentes, então melhor usar o padrão boneca russa.

Em [XFront, 2009a] são apresentados ainda exemplos considerando os três padrões e os elementos apresentados na Figura 15. O elemento *Person* tem um atributo identificador e o elemento *Book* referencia a pessoa que escreveu o livro, utilizando atributos ID/IDREF. As Figura 16, Figura 17 e Figura 18 apresentam os XSDs para os elementos da Figura 15, considerando os padrões boneca russa, fatia de salame e veneziana.

```
<Person id="rbach>
        <Name>Richard Bach</Name>
</Person>
```

```
<Book>
        <Title>Illusions</Title>
```

```
        <Author idref="rbach"/>
</Book>
```

**Figura 15 – Elementos *Person* e *Book***

No exemplo da boneca russa (Figura 16), os tipos estão todos aninhados.

```
<xsd:element name="Person">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="Name" type="xsd:string"/>
         </xsd:sequence>
         <xsd:attribute name="id" type="xsd:ID" use="required"/>
      </xsd:complexType>
   </xsd:element>

   <xsd:element name="Book">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="Title" type="xsd:string"/>
            <xsd:element name="Author">
               <xsd:complexType>
                  <xsd:attribute name="idref" type="xsd:IDREF"
                                 use="required"/>
               </xsd:complexType>
            </xsd:element>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
```

**Figura 16 –XSD para os elementos *Person* e *Book* utilizando o padrão boneca russa**

Usando o padrão fatia de salame (Figura 17), todos os componentes são espalhados, declarados individualmente, e então são agregados para formar os elementos *Person* e *Book*. Uma característica importante deste padrão é que todos os elementos e atributos são globais. Consequentemente, seus *namespaces* serão sempre expostos nas instâncias dos documentos, independentemente do valor de *elementFormDefault*.

```
  <xsd:element name="Name" type="xsd:string"/>

   <xsd:attribute name="id" type="xsd:ID"/>

   <xsd:element name="Person">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element ref="Name"/>
         </xsd:sequence>
         <xsd:attribute ref="id" use="required"/>
      </xsd:complexType>
   </xsd:element>

   <xsd:element name="Title" type="xsd:string"/>

   <xsd:attribute name="idref" type="xsd:IDREF"/>

   <xsd:element name="Author">
      <xsd:complexType>
         <xsd:attribute ref="idref" use="required"/>
      </xsd:complexType>
   </xsd:element>

   <xsd:element name="Book">
      <xsd:complexType>
         <xsd:sequence>
```

```
            <xsd:element ref="Title"/>
            <xsd:element ref="Author"/>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
```

**Figura 17 - XSD para os elementos *Person* e *Book* utilizando o padrão fatia de salame**

Com o uso do padrão veneziana (Figura 18), tipos são espalhados e declarados globais e declarações de elementos aninhadas são feitas considerando os tipos. Observe que elementos estão aninhados dentro de tipos. Isto permite habilitar/desabilitar a exposição de *namespace* nos documentos de instâncias.

```
<xsd:complexType name="Person">
      <xsd:sequence>
         <xsd:element name="Name" type="xsd:string"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID" use="required"/>
   </xsd:complexType>

   <xsd:complexType name="Author">
      <xsd:attribute name="idref" type="xsd:IDREF" use="required"/>
   </xsd:complexType>

   <xsd:complexType name="Book">
      <xsd:sequence>
         <xsd:element name="Title" type="xsd:string"/>
         <xsd:element name="Author" type="Author"/>
      </xsd:sequence>
   </xsd:complexType>

   <xsd:element name="Person" type="PersonType"/>

   <xsd:element name="Book" type="BookType"/>
```

**Figura 18 - XSD para os elementos *Person* e *Book* utilizando o padrão veneziana**

Para os exemplos acima, foi considerado que os elementos *Person* e *Book* foram declarados dentro do mesmo *namespace*. Agora. considerando que os elementos *Person* e *Book* sejam declarados em *namespaces* separados e depois importados para um esquema de catálogo, teríamos o documento de instância apresentado na Figura 19. De onde o esquema *Catalog* obteve os elementos *Person* e *Book* é totalmente transparente para o documento de instância. O conhecimento do *namespace* é escondido dentro do esquema.

```
<cat:Catalog xmlns:cat="http://www.catalog.org"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation=
                        "http://www.catalog.org
                         Catalog.xsd">
   <Person id="rbach">
      <Name>Richard Bach</Name>
   </Person>
   <Book>
      <Title>Illusions</Title>
      <Author idref="rbach"/>
   </Book>
</cat:Catalog>
```

**Figura 19 – Instância dos elementos *Person* e *Book* para padrões boneca russa e fatia de salame**

Por outro lado, o documento de instância considerando o padrão fatia de salame é preenchido com muitos qualificadores de *namespace*, como apresentado na Figura 20. Onde o esquema *Catalog* obteve os elementos *Person* e *Book* é visivelmente exposto no documento de instância, e isto é independente do valor de *elementFormDefault*. Isto ocorre porque este padrão de projeto declara todos os elementos como sendo globais. Isto demonstra a inabilidade do padrão fatia de salame esconder *namespaces*.

```
<cat:Catalog xmlns:cat="http://www.catalog.org"
             xmlns:a="http://www.person-book.org"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation=
                        "http://www.catalog.org
                         Catalog.xsd">
   <a:Person a:id="rbach">
      <a:Name>Richard Bach</a:Name>
   </a:Person>
   <a:Book>
      <a:Title>Illusions</a:Title>
      <a:Author a:idref="rbach"/>
   </a:Book>
</cat:Catalog>
```

**Figura 20 - Instância dos elementos *Person* e *Book* para o padrão veneziana**

Com o padrão Jardim do Éden obtêm-se o maior reuso dentre os padrões. De acordo com este padrão, todos os elementos e tipos são definidos no namespace global, e então referenciados elementos quando necessário. Um exemplo é apresentado na Elementos e tipos podem ser reutilizados livremente. Não existe encapsulamento de elementos neste esquema. Existem vários elementos raiz em potencial, dado que eles são globais. Este padrão pode tornar fácil a manutenção de esquemas em uma abordagem SOA, dado que ele permite recombinar aspectos dos esquemas existentes quando necessário. Este pode ser uma boa escolha a ser considerada para tipos gerais ou tipos principais em uma abordagem SOA que se espera que sejam muito reutilizáveis, ou que não estejam estrelados a um único domínio de negócio específico. Obviamente diferentes padrões podem ser utilizados simultaneamente, dependendo do nível de encapsulamento e flexibilidade necessários.

```
 <?xml version="1.0" encoding="utf-8" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            elementFormDefault="qualified">
   <xsd:element name="Name" type="xsd:string"/>

   <xsd:attribute name="id" type="xsd:ID"/>

  <xsd:complexType name="PersonType">
       <xsd:sequence>
          <xsd:element ref="Name"/>
       </xsd:sequence>
       <xsd:attribute ref="id" use="required"/>
   </xsd:complexType>

  <xsd:attribute name="idref" type="xsd:IDREF"/>

  <xsd:complexType name="AuthorType">
         <xsd:attribute ref="idref" use="required"/>
  </xsd:complexType>

  <xsd:element name="Person" type="PersonType"/>

  <xsd:element name="Title" type="xsd:string"/>

   <xsd:element name="Author" type="AuthorType"/>
```

```
   <xsd:complexType name="BookType">
       <xsd:sequence>
          <xsd:element ref="Title"/>
          <xsd:element ref="Author"/>
       </xsd:sequence>
    </xsd:complexType>

    <xsd:element name="Book" type="BookType"/>

</xsd:schema>
```

**Figura 21 - XSD para os elementos Person e Book utilizando o padrão jardim do Éden**

# 3 Modelo de Dados Canônico

Hewitt [2009] aponta que, se uma organização deseja empregar um modelo de dados canônico, é necessário que um arquiteto de dados crie este modelo seguindo uma análise detalhada dos dados da organização em uma perspectiva de gestão de dados principal, e antes de se ter muitos serviços instalados. Hewitt aponta ainda que as ações tomadas podem variar, mas a solução provavelmente irá definir esquemas locais para os serviços que reutilizam esquemas de camadas separadas, definidos independentemente dos serviços, no nível da organização.

É importante realizar uma análise e projeto de esquema em separado, antes do processo de análise de serviços. Serviços irão trocar dados do negócio, e estes dados correspondem a dados de entidades que cruzam domínios. Por exemplo, um conjunto variado de serviços através de domínios de negócio diferentes potencialmente necessitará usar certos tipos principais, tais como Cliente, Produto ou Fatura.

Existem duas escolhas mais diretas para tratar este problema:

- Maximizar reuso de tipos e eliminar definições de tipos redundantes, através da definição dos esquemas de Cliente, Produto e Fatura usando um padrão de esquema que permita recomposição. Escrever esquemas para representar os tipos principais ou mais gerais. Escrever esquemas específicos para os serviços reutilizando os tipos principais. Este é um estilo de projeto primitivo. Se um tipo principal for alterado, todos os serviços que utilizam aquele tipo deverão ser testados para verificar se continuam funcionando.
- Maximizar flexibilidade dos serviços e minimizar interdependência, definindo os tipos Cliente, Produto e Fatura separadamente para cada serviço. Dessa forma, há um contrato totalmente customizado para cada serviço. Isto pode significar muita redundância, também conhecido como desnormalização de esquema, mas permite que serviços evoluam independentemente. Serviços também ficarão totalmente encapsulados. O problema neste caso é que a quantidade de redundância pode levar a um grande impacto se um esquema de um tipo principal que é definido em vários esquemas diferentes evolui. Dessa forma, será necessária uma grande quantidade de trabalho manual com grandes possibilidades de erro para alterar e testar todos os serviços que definem o tipo de dados principal que foi alterado.

O modelo de dados canônico refere-se à prática de definir esquemas únicos para representar tipos globalmente, e reusar estes tipos referenciando-os nos esquemas

projetados localmente para o serviço. Do mesmo modo que o Enterprise Service Bus atua como um mediador, permitindo que qualquer número de web services conversem uns com os outros se ter que criar um relacionamento ponto-a-ponto específico entre cada um deles, um modelo canônico permite que sejam definidas esquemas de cópia de "ouro" para os tipos em uma organização. Qualquer serviço que necessite falar tipos pode usar o barramento de serviços ou algum mecanismo semelhante para traduzir sua versão na versão canônica.

O esquema canônico é padrão principal. Cada serviço que refere a uma entidade pode necessitar customizar o esquema canônico de alguma forma para realizar seu processamento local. Mas estas diferenças são específicas para o esquema do serviço, e qualquer esquema específico pode ser transformado para o esquema canônico. Como o esquema canônico e do serviço são esquemas distintos, maximiza-se reuso e liberdade de trabalho.

O esquema canônico não precisa ser totalmente genérico para acomodar diferenças entre esquemas de serviços distintos. Ele pode conter restrições. No entanto, ele é mais flexível e geral do que o esquema da entidade local ao serviço. Pode-se levar qualquer representação de esquema de entidade local para o esquema canônico; e, do esquema canônico, pode-se levar para qualquer outra representação. No barramento, pode-se utilizar XSLT dentro de uma orquestração ou usar um serviço de interceptação para traduzir de um tipo de dados para outro quando necessário.

Como exemplo, considere que uma empresa adquiriu outra empresa. Ambas as empresas possuem a entidade Cliente. No entanto, o esquema de ambas as entidades é diferente. Com o modelo de dados canônico, basta que os dois serviços mapeiem para o modelo canônico; eles não precisam mapear de um para o outro ou serem individualmente reescritos (o que é antagônico ao propósito de integração). Neste exemplo, têm-se três esquemas, o do cliente canônico, o esquema do serviço e o esquema que definido pela empresa adquirida. Ao longo do tempo, governança dos serviços pode ser utilizada para migrar esquemas para estarem mais de acordo com o esquema canônico a fim de cortar gastos potenciais com transformações em tempo de execução.

Hewitt [2009] advoga iniciar com o projeto de esquema como uma tarefa chave em uma abordagem SOA e apresenta as seguintes recomendações:

- Tão quanto for possível, projete esquemas para o modelo de dados canônico separadamente do projeto do serviço, antes de criar os contratos do serviço;
- Mantenha os tipos nos esquemas canônicos de alguma forma flexível e geral, dado que eles irão potencialmente absorver uma variedade de definições de entidades. Por exemplo, é provavelmente bom usar uma enumeração em um esquema de serviço, mas usar string para representar o mesmo tipo de dados no esquema canônico. Por exemplo, a enumeração de estados pode incluir os 50 estados americanos, mas os tipos enumerados do serviço que se precisa integrar incluem Guantânamo e Porto Rico além dos 50 estados americanos.
- Projete esquemas de serviços de entidades como documentos independentes que serão incluídos pelo WSDL. Não defina tipos localmente dentro do WSDL.
- Quando definir um esquema para um serviço, importe a definição do modelo canônico tão quanto for possível.

- Aplique padrões da empresa para todos os projetos de esquema, mantenha a gestão do modelo canônico com atividades de governança. Não tente centralizar tipos de entidades do negócio principais (por exemplo, Cliente ou Fatura), ao menos que se tenha uma estrutura de governança para gerir mudanças de esquema.
- Mantenha os esquemas canônicos em um repositório da organização.
- Considere definir certos tipos fundamentais que estão usados de forma central no seu próprio namespace, e os seus esquemas próprios.
- Não requeira que os esquemas canônicos sejam o *frontend* para qualquer banco de dados. Eles existem para definir conceitos e agem como mediadores ideais.
- Restrições de padrões são melhor utilizadas nos esquemas de serviço do que em esquemas canônicos. Eles podem ter um papel importante na validação.
- Evite utilizar construtos de esquema avançados, tais como, polimorfismo, escolha (*choice*) ou uniões (*unions*). Eles irão impactar drasticamente a interoperabilidade. Além disso, o suporte de ferramentas para gerar código em muitas plataformas não existe ou é muito simples.

Hewitt [2009] advoga que deve-se iniciar a partir do modelo de dados usando este modelo para compor a idéia do contrato do serviço como um facilitador para troca de dados. Adie a escrita de código tanto quanto possível. Depois que os esquemas estão claros, o contrato pode ser encaixado neles, e ambos podem ser refinados em um processo iterativo. Então, pode-se gerar código Java para os serviços serem usados, ou para os desenvolvedores completarem a implementação. Isto ajuda a garantir a não se estar atrelado a uma implementação específica porque ela imita o que seus clientes terão que fazer.

A evolução do esquema implica na evolução do contrato (pois o esquema faz parte do contrato) e na necessidade de re-gerar código. A geração de código deve ser parte do processo de construção. Objetos gerados não devem ser armazenados no repositório, e não se deve confiar em características específicas de implementação para os serviços. Trabalhar com Java pode induzir a considerar características específicas da linguagem, o que não deve acontecer para o contrato de serviço.

Não há uma regra específica. Certo grau de acoplamento ou redundância poderá ocorrer. Não é possível saber exatamente como relações entre requisitos e negócio irão evoluir e mudar ao longo do tempo.

# 4 Conclusões

Hewitt [2009] enfatiza que muitos praticantes de SOA chamam o modelo canônico de dados como um anti-padrão, e que deve-se estar muito consciente quando utilizá-lo. Outros apóiam o uso do modelo canônico de dados e, inclusive, indicam iniciar uma abordagem SOA elaborando-o. Esta é uma tarefa que utiliza ontologia [Gruber, 2008] e taxonomia.

O modelo canônico não é um anti-padrão se são realizados grandes esforços para análise e esclarecimento de conceitos. Esteja certo de incluir tudo o que uma entidade pode precisar dentro da organização, e mantenha isto geral o suficiente. Use um padrão de esquema que permita flexibilidade no projeto.

Então, quando você definir os serviços de entidade, defina o esquema local para o serviço. Reutilize tipos de dados do modelo canônico tão quanto o possível, via importação (*import*).

Qualquer mudança no modelo canônico deve ocorrer através da governança SOA, e elas devem ser raras. Serviços representam entidades do negócio. Estas coisas não mudam todo o dia. Será necessário realizar mudanças no modelo canônico eventualmente. Neste caso, pode-se versionar estes esquemas explicitamente, e se a versão for indicada no *namespace*, têm-se liberdade para mudar apenas o serviço que se deseja mudar sem ter que alterar todos os serviços de uma vez. Atualize os demais serviços ao longo do tempo, mas não demore em realizar esta atualização. Ter mais de duas versões de esquema canônico, pode levar a não se ter mais o esquema canônico. Defina atividades de governança para este processo.

# 5 Referências


Gruber, T.R. **Ontology**. In: Liu, L. and Özsu, M. (Eds.) Encyclopedia of Database Systems, Springer-Verlag, 2008.

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

JOSUTTIS, N. M. **SOA in practice: The Art of Distributed System Design**, O'Reilly, 2007.

OASIS. **Web Services Business Process Execution Language (WS-BPEL)**. Organization for the Advancement of Structured Information Standards (OASIS), 2007. Disponível em <http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel>. Acessado em 5 Dez. 2009.

W3C. **Extensible Markup Language (XML) 1.0 (Fifth Edition).** World Wide Web Consortium (W3C), 2007. Disponível em <http://www.w3.org/TR/REC-xml/>. Acessado em 5 Dez. 2009.

XFront. **Global versus Local.** XFront tutorials and articles on XML and Web Technologies. Disponível em <http://www.xfront.com/GlobalVersusLocal.html#FirstDesign>. Acessado em 5 Dez. 2009a.

XFront. **Should it be an Element or a Type?** XFront tutorials and articles on XML and Web Technologies. Disponível em <http://www.xfront.com/ElementVersusType.html>. Acessado em 10 Dez. 2009b.

W3C. **XML Schema Part 0: Primer Second Edition. .** Wide Web Consortium (W3C), 2004. Disponível em <http://www.w3.org/TR/xmlschema-0/#PO>. Acessado em 6 Dez. 2009.

W3C. **XML Schema Part 1: Structures Second Edition.** Wide Web Consortium (W3C), 2008. Disponível em <http://www.w3.org/TR/xmlschema-1/>. Acessado em 5 Dez. 2009.

W3C. **Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language.** World Wide Web Consortium (W3C), 2007. Disponível em <http://www.w3.org/TR/wsdl20/>. Acessado em 5 Dez. 2009.

W3Schools. **XML Schema Tutorial**. W3Schools.com. Disponível em <http://www.w3schools.com/schema/default.asp>. Acessado em 6 Dez. 2009.