

# Acoplamento entre serviços SOA

## SOA e a redução do acoplamento

**Quais os benefícios da redução de acoplamento em WebServices em uma arquitetura orientada para serviços, e porque devemos buscá-la ?**

GLAUCO REIS

O baixo acoplamento é provavelmente um dos benefícios mais discutidos em TI de uma forma geral, mas principalmente quando o assunto são WebServices, bem ao lado da reutilização de serviços. Hoje, após uma extensiva utilização dos princípios de SOA, será que podemos medir este desacoplamento ? Ou, será que sabemos como aplicá-lo ?

Vamos discutir neste artigo um pouco sobre o acoplamento, como as ferramentas atuais para implementação de SOA e BPM propiciam a criação de códigos pouco acoplados e alguns cuidados que devemos ter na seleção de nossa arquitetura de implementação.

### O que é acoplamento?

Bem, de uma forma bastante ampla, o acoplamento mede o grau de dependência entre dois sistemas ou módulos. Este conceito pode ser aplicado de forma semelhante a serviços. Quanto maior é o acoplamento, maior é a dependência entre o serviço e o cliente que o acessa, e portanto alterações no serviço irão demandar manutenções em maior ou menor grau em outras partes do sistema. Isto pode ser traduzido em uma palavra como “**manutenção**”. Naturalmente, o desejável é que o serviço e seu cliente estejam o mais desacoplados possível, de forma que alterações no serviço causem o menor impacto no restante dos códigos em um sistema.

Parece óbvio que desacoplamento total parece impossível de se obter, já que para que o cliente tenha como acessar um serviço precisa no mínimo conhecê-lo, em termos de parâmetros passados e retornados. Acoplamento pode existir de outras formas também. Imagine duas rotinas que fazem uma determinada operação, com ligeiras diferenças. Por uma deficiência na implementação estes dois códigos foram criados de forma separada. Caso exista a necessidade de alteração em uma rotina, a outra deverá também ser alterada (para manter os processamentos idênticos). Este é um tipo de acoplamento que pode ser reduzido mantendo-se apenas um ponto de manutenção para um determinado código. Como existem inúmeros conceitos onde o acoplamento pode ser aplicado, vamos discutir o acoplamento aplicado a WebServices e os clientes que os acessam, e naturalmente os desdobramentos que a evolução das tecnologias sofreram durante esta evolução.

### Acoplamento, CORBA e RMI?

Em um mundo onde a internet convive com múltiplos sistemas operacionais, o termo acoplamento ganhou novas proporções. Diz-se que a necessidade de uma tecnologia em um ambiente para que seja possível acessar outro ambiente é um tipo de acoplamento. Por exemplo, termos o Windows instalado na máquina cliente para que possamos acessar um serviço **DCOM** em outra máquina é um tipo de acoplamento. Da mesma forma, a necessidade de uma **JVM** para acessar um serviço **JRMI** em outra máquina é um tipo de acoplamento. Já **CORBA** (Common Object Request Broker Architecture), como agregou um protocolo aberto para comunicação (**IOP** – **Internet InterOrb Protocol**) tem um mais baixo acoplamento em relação aos anteriores, já que existem provedores CORBA para diversos sistemas operacionais e mesmo linguagens de programação existentes. Nesta escala de acoplamento, a **DCOM** ficaria em um extremo, ao passo que o **CORBA** em outro. Com a popularização do XML como forma de representação de informações de objetos, foi até que natural a criação de um protocolo que utilizasse dados codificados em XML como forma de comunicação. Na realidade, quem acompanhou a criação do SOAP sabe que ele foi criado como forma de permitir

interoperabilidade entre as plataformas Microsoft e JAVA. Um dos criadores do SOAP foi a própria Microsoft, permitindo que seus sistemas pudessem conversar de forma “neutra” com outras tecnologias como o JAVA. Como foi definido sobre padrões abertos em grande ascensão (**HTTP** e **XML**), teve rápida resposta positiva por parte do mercado.

É bem verdade que a maioria dos problemas de comunicação entre os vários sistemas já estavam resolvidos com o IIOP, bem antes do advento do protocolo SOAP, haja vistas que em 1995 o CORBA já era estável e o SOAP só foi criado por volta de 1999 (com a versão 1.1 lançada em 2001). Talvez a “mal falada” (ou malfadada?) dificuldade de programação do CORBA tenha sido a principal causa de seu sepultamento, já que tecnicamente ele é muito superior ao SOAP atual e vários dos problemas que estão sendo endereçados agora já estavam resolvidos de forma eficiente no CORBA (segurança e transação, por exemplo).

Em termos de acoplamento, as tecnologias citadas anteriormente dependem de um código que é gerado a partir do serviço, normalmente chamado de “stub”. Este “stub” implementa o pattern proxy do GOF, fazendo com que o cliente não tenha a real noção de onde o serviço está sendo executado, se na mesma máquina ou em qualquer lugar da rede. Esta é a chave para os serviços distribuídos. Se por um lado permite que o código esteja em qualquer lugar da rede, por outro força o cliente a agregar o stub, para fornecer esta transparência. É exatamente este stub que traz um maior grau de acoplamento entre o consumidor do serviço e o produtor.

## WebServices e SOAP

A idéia do SOAP (Simple Object Access Protocol) foi a princípio muito simples. Seria proporcionar a comunicação entre diversas tecnologias orientadas para Objeto (CORBA, RMI, DCOM, etc), utilizando um mecanismo padronizado de comunicação. Todos os objetos seriam transcritos para XMLs, e através de uma requisição HTTP (que passaria pela internet com tranquilidade) as informações seriam passadas ao servidor. Do lado servidor existirá um programa “escutando” HTTP, que receberia a requisição (esta também em formato XML) e transfere os dados em XML para a tecnologia nativa. O retorno (um XML) seria enviado de volta ao cliente. Através de uma jogada que poderia ser considerada mais de marketing do que propriamente tecnológica, o SOAP se tornou WebService. Além de agregar dois outros protocolos (WSDL e UDDI), estrategicamente retirou o termo “Object” (a letra O de SOAP) e acrescentou o termo “Services” (em WebServices). Desta forma, parece que um serviço pode ser utilizado de forma mais genérica, sem estar presa a uma linguagem orientada para objetos. Embora isto seja verdadeiro na prática, pois serviços podem ser chamados e criados em linguagens não OO, o envio das informações ainda é feito seguindo as regras de um XSD, que é orientado para objetos. Cabe à tecnologia não OO tratar as particularidades dos objetos, transformando-os em algo que pode ser lido de forma estruturada. Esta transformação em XML das informações já reduziu o acoplamento. Pode-se dizer que um serviço codificado em SOAP tem menor acoplamento do que um criado em CORBA. Uma API que permite a chamada do serviço de forma dinâmica (JAXP – Java for XML Processing) foi criada, fazendo com que possamos chamar um serviço sem que precisemos do “stub” do lado cliente. Embora não necessitemos de JAXP, já que o serviço pode ser chamado por qualquer tecnologia que permita montar um XML, ele simplifica a geração do XML que executa o serviço.

## Passagem de dados para os serviços

Quase sempre a chamada ao serviço implica na passagem de alguns parâmetros e no tratamento dos dados de retorno. Com uma frequência também grande os dados precisam ser manipulados por uma linguagem de programação OO como o Java, e não é prático manipular o retorno dos dados em formato XML. As duas tecnologias básicas para leitura de XML são o SAX e o DOM.

O SAX faz a leitura varrendo o arquivo e buscando as informações desejadas, implementando um pattern Visitor do GOF. Isto não resolve pois não sabemos quais informações iremos necessitar. Outra forma seria utilizar DOM (Document Object Model), que lê um XML para a memória como uma árvore de Objetos, mas deixa muito a desejar em termos de praticidade, pois gera os objetos de uma forma não orientada para a linguagem JAVA (**Veja o quadro Problemas com o DOM**).

Na tentativa de resolver estes problemas, foi criada uma tecnologia chamada de XML Binding, que teria como objetivo a transformação de um XML (ou mais precisamente um Schema) em uma

estrutura de classes muito mais fácil de manipular em Java. Infelizmente, como veremos, o XML Binding fez com que os serviços aumentassem o acoplamento.

## XML Binding e acoplamento

A idéia do XML Binding foi a de criar uma API que cuidasse de toda complexidade envolvida no parseamento do XML e dos dados para seus tipos respectivos, transformando um XML em uma estrutura Java genuína, e vice versa. É como que uma ponte entre o mundo XML e SOAP para o mundo Java (ou outra linguagem). Não existe dúvida que realmente facilita o acesso. Atualmente, praticamente todas as soluções de mercado utilizam uma tecnologia para XML binding, sendo as mais divulgadas o JAXB ou o XMLBeans. O XMLBeans é mais antigo e foi desenvolvido pela BEA, e posteriormente doado ao Apache GROUP.

O JAXB seguiu na esteira de sucesso do XMLBeans, e embora não seja tão estável têm muito mais documentação no mercado, sendo liderada pela Sun. Esta discussão é quase que uma religião, e existem defensores ferrenhos do XMLBeans e o mesmo com o JAXB. Para nossa discussão, eles operam de forma similar (**Veja os quadros Utilizando XMLBeans e Utilizando JAXB**). O que nos importa é que as duas APIs geram classes Java que são compiladas mantidas compiladas e que permitem criar instâncias Java a partir do XML (**UnMarshalling**) e o mesmo para gerar o XML a partir das instâncias Java (**Marshalling**).

Embora esta abordagem gere grande simplicidade para o programador, vale lembrar que o código para **Marshalling** e **UnMarshalling** provavelmente estarão máquinas diferentes. Isto significa dizer que utilizar um XML Binding como o **JAXB** ou **XMLBeans** normalmente traz um grau maior de acoplamento em favor de uma facilidade maior no acesso às informações.

Se fizermos um comparativo com o passado, é engraçado que uma das maiores críticas sobre o JRMII, CORBA e EJB 2.X era exatamente a geração dos “**stubs**” e “**skeletons**” (códigos Java estáticos), paradigma este que foi quebrado pelo **JBOSS** (que gera os stubs de forma dinâmica) e duramente criticado como sendo lento pela maioria dos fabricantes de servidores de aplicação. O momento atual parece favorecer a utilização de tecnologias de XML binding, trazendo ao programador Java um tratamento da informação SOAP mais transparente.

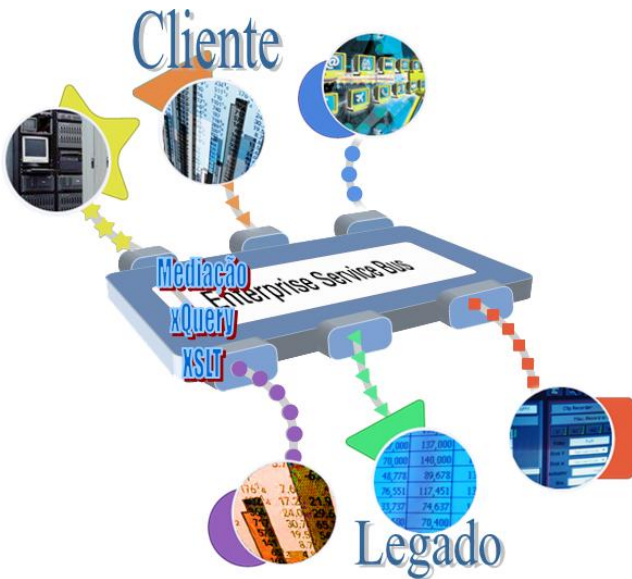
As várias soluções tratam o problema de formas distintas. Algumas soluções tratam as informações gerando códigos estáticos e outras o fazem dinamicamente. Por exemplo, na suíte da Oracle/BEA, o WLI (suíte de integração) utiliza XMLBeans gerados estaticamente, forçando recompilação e publicação a cada mudança de modelo de dados. Por outro lado, o ALSB lida de forma dinâmica com o tratamento do XML (não transparece sua implementação para o programador). Em eventuais mudanças nas informações passadas aos serviços, o WLI tende a gerar mais manutenção do que o ALSB. Nos pontos onde o acoplamento acontece de forma estática (mais fortemente acoplados) tendemos a ter mais demandas por manutenção. Se o arquiteto consegue identificar estes pontos nas fases iniciais do projeto, pode minimizar os impactos de manutenção causados pelo acoplamento mais alto, criando a arquitetura ideal para o projeto.

## Como um ESB reduz o acoplamento ?

Agora, imaginemos uma implementação em um contexto mais amplo, corporativo e com centenas ou mesmo milhares de serviços seguindo boas práticas como a adoção de um modelo canônico (**Veja o quadro Modelos corporativos ou canônicos**). Utilizando uma arquitetura JEE a menor granularidade encontrada é o WAR (alguns servidores definem o menor grânulo como um EAR). Em SOA, o elemento mais comumente encontrado é o serviço. Mas um WAR ou um EAR contém muitas centenas de serviços. Em resumo, gerenciar um ambiente JEE tende a ser mais simples do que gerenciar um ambiente SOA, já que em um ambiente SOA o número de blocos é muitas vezes maior. Qualquer empresa que já tenha implementado projetos em SOA sabe que esta frase é especialmente verdadeira. Precisamos de uma ferramenta que permita gerenciar os serviços de uma forma centralizada, operando como um barramento de controle dos serviços. Este conceito foi inicialmente introduzido por David Chappell é chamado de ESB (Enterprise Service Bus). Uma solução ESB é reconhecida como uma ferramenta essencial na implementação de qualquer projeto SOA.

Um dos papéis do ESB é mediar a comunicação entre os serviços. Imagine o seguinte cenário : um serviço é criado para atender uma demanda em um sistema. Após algum tempo, um novo serviço é necessário, mas com parâmetros ligeiramente diferentes do serviço já criado e sendo utilizado. Se a comunicação tiver alto acoplamento, quando a nova assinatura for implantada deve forçar um ciclo de manutenções para todos os clientes que utilizam o serviço. Uma forma de reduzir o acoplamento em ESBs é manter pelo menos duas camadas de serviços. Uma camada de mais alto nível interage com o cliente, e uma camada de mais baixo nível interage com o legado. A camada de mais alto nível é “roteada” para a camada de mais baixo nível.

**Figura Q1.** Função de um ESB



Quando um serviço é “roteado” através do ESB, além de reduzirmos o acoplamento, já que a chamada não é direta, ainda podemos resolver o problema de múltiplos parâmetros descrito anteriormente para um mesmo serviço, pois durante o roteamento podemos fazer transformações nos dados, utilizando XSLT ou xQuery (**Ver quadro XSLT e xQuery**). Ou seja, durante o roteamento os dados na entrada podem ser transformados em outro formato para a chamada do legado. Algumas soluções permitem além de transformações muito complexas a composição de serviços, ou seja, permitem a criação de outros serviços utilizando os já existentes através de mediações e transformações. Em um ambiente realista SOA um ESB é uma peça que deve ser seriamente considerada. A solução ESB ideal é aquela que permite um baixo acoplamento, sempre que possível fazendo as mediações e transformações de forma dinâmica, para evitar recompilações devido ao XML binding descrito acima.

## Conclusões

Bem, olhando as suites de empresas líderes de mercado como IBM e Oracle, percebemos que estamos cada vez mais produzindo quantidades menores de códigos. O que conhecemos como programas estão sendo orquestrados por XMLs (BPEL e XPD), transformados por XMLs (XSLT e xQuery), e os códigos Java, quando aparecem, estão próximos ao legado como acesso a tecnologias de integração (se tornaram serviços). O objetivo disto tudo é depender cada vez menos de ciclos de compilação complexos e demorados, aumentando o dinamismo na criação de aplicações corporativas. Isto se baseia no fato de que publicar arquivos XML tendem a ser extremamente mais fácil do que compilar e publicar códigos Java. Este dinamismo somente pode ser conseguido quando tornamos estes XMLs mais flexíveis e adaptáveis. Da mesma forma que em Java, neste mundo XML podemos criar tanto programas altamente acoplados e de difícil manutenção como códigos desacoplados e que agilizam a manutenção. O fato de utilizarmos estas soluções não garantem que teremos a agilidade esperada. Entender a solução, quais seus pontos de

difícil manutenção e como podemos criar códigos com máximo de reaproveitamento são a chave do sucesso para a próxima geração de aplicações do futuro.

## Problemas com o DOM

Uma das formas de se transformar as informações em Objetos é montando-se uma árvore DOM com o XML de retorno. Esta API é muito antiga e tende a ser estável e muito rápida, entretanto o retorno gerado não têm uma aparência de instâncias Java, pois a árvore gerada é acessada como se fosse uma árvore, com uma navegação recursiva, onde Objetos do tipo **Node** contém outros objetos do tipo **Node**, e por aí vai. Para uma navegação precisa, quase sempre você precisa se preocupar com nomes como **child** (se o elemento é filho), **sibling** (se está no mesmo nível) ou **parent** (está em um nível superior). Veja alguns dos métodos para navegação em um **Node** :

```
Node getChild();
Node getFirstChild();
Node getLastChild();
Node getNextSibling();
Node getPreviousSibling();
Node getParentNode();
NodeList getChildNodes();
```

Os **Nodes**, que representam os elementos de mais alto nível podem ser do tipo **Element** (representa um elemento do XML), **Attribute** (atributos também são representados como nós da árvore!), **Text** (o que estiver como informação dentro de uma abertura e fechamento de dados ou **Document** (o próprio documento XML).

Outro problema é que Quando você chega na informação, têm um Objeto do tipo **Text**, que aponta para o conteúdo. Mesmo tipos como **double**, **integer**, **date** são tratados como **Strings**, e devem ser convertidos no momento em que se necessita a informação (lembre-se de que erros podem acontecer na transformação). Isto é muito improdutivo, e várias vezes nem mesmo sabemos o tipo que estamos lendo.

Estas características fazem com que a utilização de DOM para leitura seja muito improdutivo, embora completamente desacoplado dos dados sendo lidos.

### Listagem Q1. Exemplo de parseamento via DOM

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import java.io.File;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class LeitorDeXML {
    public static void main (String args[]) {
        File docFile = new File("arquivo.xml");
        Document doc = null;
        try {
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            DocumentBuilder db = dbf.newDocumentBuilder();
            doc = db.parse(docFile);
        } catch (Exception e) {
            System.out.print("Problemas lendo o arquivo.");
        }
        Element root = doc.getDocumentElement();
    }
}
```

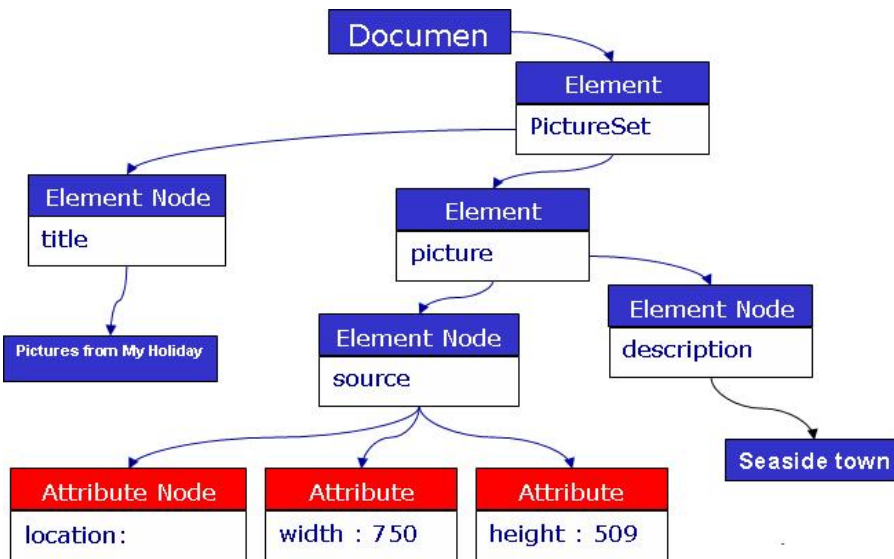
### Listagem 2. Exemplo de XML

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

```

<PictureSet>
  <title>Pictures from My Holiday</title>
  <picture>
    <source location="town.jpg" width="750" height="509"/>
    <description>Seaside town</description>
  </picture>
</PictureSet>
    
```

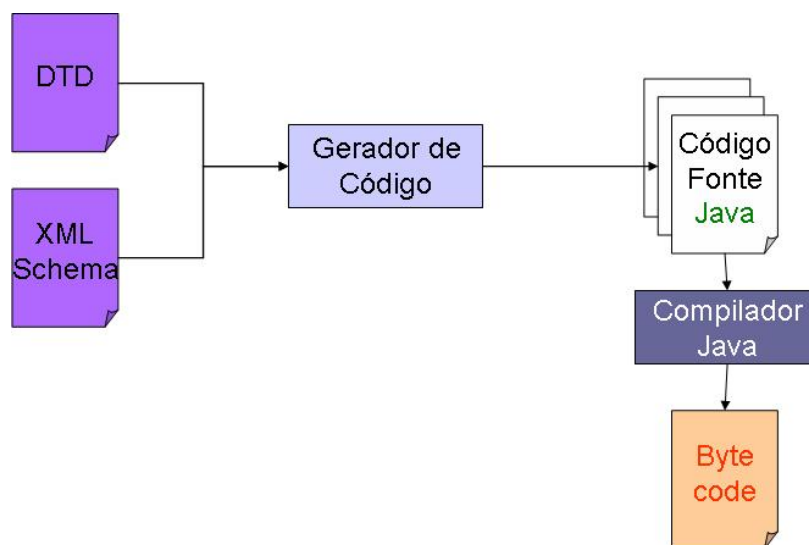
Figura Q1. Árvore gerada pelo XML acima



## Utilizando JAXB

A idéia do JAXB, assim como o XMLBeans é de uma API que gere códigos fontes do Java a partir de um arquivo XML.

Figura Q2. Arquitetura JAXB



Para gerar os códigos Java precisamos de algo que descreva o formato dos objetos. Este “descriptor” pode ser um DTD ou um Schema XSD (praticamente todas as implementações estão colocando as DTDs em um papel secundário). A partir do Schema, pode-se executar uma linha de

comando que gera as classes Java, e já que um schema é bastante formal, o Java gerado leva em consideração o nome do campo, tipos etc. A chamada via linha de comando poderia ser :

```
java -jar [jacob_dir]/lib/jacob-xjc.jar <arquivo_schema>.xsd
```

Para o schema a seguir, por exemplo :

### Listagem Q3. Exemplo de XSD (Schema)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="item">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="priority" type="xs:int"/>
        <xs:element name="task" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

O código gerado seria algo como :

### Listagem Q4. Código gerado pelo XSD anterior

```
public interface ItemType {
    java.lang.String getTask();
    void setTask(java.lang.String value);
    int getPriority();
    void setPriority(int value);
    java.lang.String getName();
    void setName(java.lang.String value);
}
```

Perceba que os getters e setters são mantidos, bem como todos os tipos dos dados. Parsear um XML para que seja gerada uma estrutura de classes é simples, e pode ser feita com o código :

### Listagem Q5. Gerando a estrutura de classes a partir do XML com JAXB

```
JAXBContext jacobContext = JAXBContext.newInstance("pacote");
Unmarshaller unmarshaller = jacobContext.createUnmarshaller();
File file = new File("item.xml");
Item item = (Item) unmarshaller.unmarshal(file);
```

O processo é bem mais simples e podemos utilizar os mecanismos do Java para alterar os campos e depois gerarmos novamente o XML a partir da estrutura de classes.

## Utilizando XML Beans

O XMLBeans também gera códigos Java que podem ser preenchidos com o conteúdo de um XML. Ele também tem um compilador de linha de comando, com uma sintaxe que define o arquivo de saída e o XSD a ser compilado.

```
scomp -out arquivo.jar item.xsd
```

### Listagem Q6. Gerando a estrutura de classes a partir do XML com XMLBeans

```
File inputFile = new java.io.File("arquivo.xml");  
ItemDocument doc = ItemDocument.Factory.parse(inputFile);  
ItemDocument.Item item = doc.getItem();
```

O processo é similar, e uma classe de Parser (**ItemDocument**) tem um método que permite obter um elemento do tipo **Item** (inner class). Depois é só utilizar os getters e setters como uma classe normal.

## Modelos corporativos ou canônicos

Uma das boas práticas já utilizadas pelas tecnologias de integração no passado, e largamente utilizadas como prática hoje em dia em projetos SOA é a adoção de um modelo corporativo ou modelo canônico. Este seria um modelo com os dados relevantes para o negócio. Os relacionamentos entre as entidades indicam as regras mais básicas do negócio. É algo normalmente corporativo e utilizado em todos os projetos da empresa. Este modelo deveria ser definido antes mesmo de uma implementação profunda de SOA. Eles poderiam ser descritos em termos de XSDs (Schemas) ou classes Java, o que normalmente não é muito relevante. Mas para que serve este modelo ?

A argumentação é simples e concisa. Imagine um ambiente corporativo com milhares de serviços. Cada um destes serviços têm uma assinatura, com parâmetros de entrada e parâmetros de saída. Diversos projetos na empresa estão em andamento e novos serviços estão sendo definidos e criados a todo momento. Por mais que tenhamos um documento com boas práticas para geração de nomes das informações que serão passadas aos serviços (algo que deveria ser obrigatório), e mesmo que tenhamos uma área de governança forte e ativa (que centralize as criações dos serviços) sempre haverá a possibilidade de colisões na definição dos serviços. Uma colisão é a definição de um serviço que faz a mesma coisa que outro.

Quando utilizamos um modelo canônico central, e forçamos a que todos os serviços recebam ou retornem alguns dos elementos do canônico, torna-se uma prática a busca pelas entidades necessárias, cada vez que um novo serviço está sendo criado. Como todos utilizam os mesmos elementos, a tarefa de encontrar serviços que estejam em colisão tende a ser mais fácil, já que eles devem ter os mesmos elementos do canônico passados como parâmetros e provavelmente o mesmo retorno. Funciona como se tivéssemos um repositório centralizado de todas as classes de entidade (entity) de um sistema Orientado para Objetos e as consultássemos a todo momento da criação ou alteração de uma classe de controle (control).

Entretanto, como um ponto centralizado de definição de todos os elementos, caso mudanças sejam feitas no modelo o impacto em termos de refatoração dos serviços pode ser considerável. Por isto uma governança sobre o modelo e quais serviços são impactados por ele se faz necessário, bem como a adoção de uma arquitetura que minimize a necessidade por mudanças.

## XSLT e xQuery

A tecnologia XSLT (XSL Transformations) foi criada com o objetivo de transformar arquivos XML de um formato para outro, e teve sua mais notada aplicação na geração de informações para apresentação (por exemplo transformar um XML em HTML). Entretanto, é uma tecnologia que permite transformações genéricas. No caso de um ESB, ela pode ser utilizada para fazer transformações de informações entre chamadas de serviços. Ou seja, ao invés de criar um código

Java para retirar uma informação e passar para um serviço, podemos criar um XSLT que irá fazer a mesma atividade. Como ele não exige recompilação, pois é um arquivo XML, facilita a atividade de gerenciamento dos serviços e ainda reduz o acoplamento. Tanto no caso do XSLT como no caso do xQuery outras tecnologias são utilizadas, como XPATH e namespaces. Em XSLT normalmente estruturamos o documento em seções (templates) que são responsáveis por tratar um elemento específico. Conforme o documento vai sendo navegado, quando o elemento descrito no template é encontrado, a regra dentro do template é aplicada. Regras podem chamar explicitamente outras regras, e no final do processamento obtemos um novo documento XML (ou mesmo outro formato) que pode ser utilizado para qualquer finalidade.

#### Listagem Q7. Exemplo de XSLT

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="name">
    <xsl:value-of select="last_name"/>-<xsl:value-of select="first_name"/>
  </xsl:template>
  <xsl:template match="person">
    <xsl:apply-templates select="name"/>
  </xsl:template>
</xsl:stylesheet>
```

#### Listagem Q8. Exemplo de XML que pode ser aplicado ao XSLT da listagem Q7

```
<?xml version="1.0" ?>
<?xml-stylesheet href="pessoa.xml" type="text/xsl" ?>
<people>
  <person born="1912" died="1954">
    <name>
      <first_name>Alan</first_name>
      <last_name>Turing</last_name>
    </name>
  </person>
</people>
```

No exemplo Q7 de XSLT, temos dois templates, um para o elemento do documento “name” e outro para o elemento “person”. O template para person faz uma chamada ao elemento name, como se fosse uma subrotina em java. O elemento name utiliza a tag **xsl:value-of** que obtem um valor no XML de entrada. Neste caso os campos **last\_name** e **first\_name** estão sendo concatenados em uma única string com um hífen entre eles. No caso de XSLT sempre existe um único XML de entrada. Ele pode ser definido dentro do XML, como foi feito na listagem Q8, ou pode ser feito externamente, via passagem de parâmetros ao parser XSLT. Existem vários “comandos” além de **xsl:value-of**, como comandos para ordenar os elementos, iterar em uma lista e selecionar um dos elementos (choice). Embora a sintaxe seja muito simples, transformações muito complexas podem ser feitas no XML fornecidos. Existem vários parsers para XSLT, inclusive vários open source como o XALAN, da apache.

Já no caso da xQuery, é uma linguagem que também proporciona transformações em documentos, mas têm uma sintaxe mais próxima de um script do que o XSLT, que é XML puro. Para o programador Java, tende a ser ligeiramente mais simples programar em xQuery.

O mesmo exemplo descrito em XSLT poderia ser descrito em xQuery como :

```
for $x in document("pessoal.xml")/people/person/name
return data($x/last_name) - data($x/first_name)
```

O xQuery tende a ter uma sintaxe mais resumida. A estrutura principal de construção do xQuery é o FLWOR (lê-se flower e significa F-for L-let W-where O-order R-return). É um for que permite filtrar em um único comando elementos, ordena-los e retorná-los. A discussão se a utilização de xQuery é mais apropriada ou XSLT também é algo religioso (como quase tudo em Java), além disto, alguns fabricantes centralizam suas implementações em xQuery (BEA) e outros focam em XSLT (IBM). Alguns autores indicam que XSLT é mais apropriado para transformações de documentos completos, enquanto que xQuery pode ser aplicado a “pedaços” de XML. Em termos de parsers xQuery a diversidade é muito menor, sendo que temos apenas uma implementação open source, e ainda é limitada, pois não implementa namespaces.

## Links

[http://en.wikipedia.org/wiki/Coupling\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Coupling_(computer_science))

Definição de acoplamento e coesão

<https://jaxb.dev.java.net/>

Implementação de referência do JAXB

<http://www.ibm.com/developerworks/library/x-xmlbeans/>

Interessante artigo da IBM sobre XMLBeans

<http://www.ibm.com/developerworks/xml/library/x-beans1/17/11/2008>

Tutorial sobre como utilizar o XMLBeans

<http://www.w3schools.com/xquery/default.asp>

Tutorial xQuery

<http://www.davidchappell.com/blog/2005/12/on-defining-esb>

Blog do David Chappell com definição de ESB

<http://saxon.sourceforge.net/>

Implementação de xQuery e documentação

## Livros

*Design Patterns, Erich Gamma-Richard Helm-Ralph Johnson-John Vlissides, Addison Wesley*

Referência aos patterns comentado neste artigo

*Enterprise Integration Patterns, Gregor Hohpe, Addison Wesley, 2003*

A referência para os padrões que estão agora sendo aplicados ao SOA

*xQuery, Priscilla Wamsley, O'Reilly, 2007*

Excelente livro sobre xQuery

**Glauco Reis** ([glauco@portalbpm.com.br](mailto:glauco@portalbpm.com.br)) é gerente de soluções SOA na Altran CIS, uma empresa com foco em SOA e BPM. Atua há mais de 25 anos em TI, com mais de 150 artigos publicados em diversas revistas nacionais, e participação em eventos como COMDEX e FENASOFT. É especialista em SOA e BPM, tendo atuado na construção de uma solução BPMS nacional, além de ser articulista do site PortalBPM ([www.portalbpm.com.br](http://www.portalbpm.com.br)). Também mantém um site pessoal explorando os temas OO, UML, SOA e BPM ([www.glaucoreis.com.br](http://www.glaucoreis.com.br)). É especialista em soluções IBM, com mais de 5000 horas de treinamento ministrado sobre SOA, BPM, Java, OO, UML e tecnologias relacionadas.