

SOA Foundation

Classic
and Next Generation

ÍNDICE

PÁGINA

	PÁGINA
1.0 INTRODUÇÃO AO SOA	1
1.2 Promessas SOA	2
1.3 Same old architecture.....	3
1.4 Como os serviços encapsulam a lógica	4
1.5 Como os serviços são desenhados	5
2.0 Desafios do mundo corporativo	7
2.1 Princípios chave da orientação por serviços	8
2.2 SOA como arquitetura distribuída	9
2.3 Soa sob ótica de negócios	10
2.4 SOA Services 143.0 WebServices	11
3.0 WebServices	15
3.1 WebServices funcionando	22
3.2 Operações Síncronas.....	23
3.3 Operações Assíncronas	24
3.4 Exercício, construção de WebServices	25
3.5 SOAP - Simple Access Protocol.....	25
3.6 Provedores de WebServices	28
3.7 Distribuição das requisições.....	34
3.8 WebServices sozinhos não implicam em SOA.....	37
4.0 SOA Stack – Infraestrutura.....	38
4.1 Workflow e Orquestração	38
4.2 SOA Management	39
4.3 SOA Governance – Registry.....	40
4.3.1 Public & Private Service Registry	41
4.4 SOA Service Bus	43
4.5. História do ESB	43
4.5.1 Arquitetura Desacoplada	44
5.0 Como definir a granularidade do serviço	47
5.1 Exercício de Modelagem	50
5.2 DataCentric	50
5.3 Modelagem de Serviços e Contratos WSDL	53

ÍNDICE

PÁGINA

6.0 Namespaces.....	55
6.1 Types.....	56
7.0 Domain-Driven Design.....	57
7.1 Model.....	59
7.2 Ubiquitous Language	60
7.3 Arquitetura em Camadas	61
7.4 Patterns	62
8.0 XML e DTD Fundamentos	65
8.1 XML Origem	66
8.2 Características do XML.....	67
8.3 Estrutura do XML	67
8.4 Estrutura de uma árvore XML.....	68
8.5 Declarações de Tipos de Elementos DTD	69
8.6 Declarações de Listas de Atributos DTD	71
8.7 Declarações de Entidades DTD	73
9.0 XML Schemas – XSD.....	80
9.1 Referenciando um Schema em um XML	82
9.2 XSD Elemento simples.....	83
9.2.1 O que é um elemento simples	83
9.2.2 Como definir um elemento simples	83
9.2.3 Tipos de dados XML Schema comuns	84
9.2.4 Declare valores padrão e fixos para elementos simples	84
9.3 Atributos XSD	85
9.3.1 O que é um atributo?	85
9.3.2 Como definir um atributo	85
9.3.3 Tipos de dados comuns do XML Schema	85
9.3.4 Declare valores padrão e fixo para atributos	86
9.3.5 Criando atributos opcionais e obrigatórios	86
10.0 Restrições de Conteúdo	87
10.1 Restrições/Facets XSD.....	87
10.2 Restrições em valores	87

ÍNDICE

PÁGINA

	PÁGINA
10.3 Restrições	87
10.4 Restrições em Séries de Valores	88
10.5 Outras restrições em séries de valores	90
10.6 Restrições em caracteres vazios	92
10.7 Restrições de comprimento	93
10.8 Restrições para tipos de dados	95
11.0 Elementos Complexos	96
11.1 Elementos Complexos Vazios.....	99
11.2 Tipos complexos apenas elementos	100
11.3 Elementos Complexos apenas textos.....	102
11.4 Elementos Complexos Mistos	103
11.5 Indicadores de tipos complexos.....	104
12.0 O Elemento <ANY>	110
12.1 O Elemento <ANYATTRIBUTE>.....	111
12.2 Substituição de Elementos.....	113
12.3 Elementos	115
13.0 Xpath Conceitos.....	136
13.2 Localizando Nós	138
13.3 Location PATHS - Caminhos Locais.....	141
13.4 Expressões XPATH.....	148
13.5 Funções XPATH	150
13.6 Exercícios XPATH.....	153
14.0 Xquery FastTrack	155
14.1 Como Selecionar Nós	157
14.2 Expressões FLWOR("Flower")	158
14.3 XQuery Sintaxe.....	159
14.4 Expressões Condicionais	159
14.5 Comparações	160



ÍNDICE

PÁGINA

14.6 Adicionando Elementos e Atributos	160
14.7 Seleccionando e Filtrando Elementos.....	162
14.8 XQuery Functions.....	164
15.0 SOA Security	166
15.1 Security Specification and Standards	167
15.2 Transport-Level Security	168
15.3 Security Specification and Standards	168
15.4 WS-Security.....	169
15.5 Usando criptografia XML	173

1 INTRODUÇÃO AO SOA

"A service-oriented architecture (SOA) is the organizational and technical framework that enables an enterprise to deliver self-describing, platform-independent business functionality and make it available as building blocks of current and future applications." - Carl August Simon

Princípios da arquitetura SOA

Para as companhias há muita promessa e expectativa em cima da arquitetura SOA, como, por exemplo, responder mais rápido às mudanças de negócio e otimizar o número de desenvolvimento de "serviços (rotinas)" redundantes em plataformas heterogêneas.

Dentre os benefícios chave:

- Oferecer uma melhor produtividade, agilidade e velocidade tanto para a área de negócios quanto TI.
- Permitir que a equipe de TI desenvolva serviços em alinhamento às expectativas do negócio.
- Excelente tempo de resposta, melhorando a experiência do usuário final do software .
- Encapsular a complexidade tecnológica de integrações entre as mais diferentes plataformas da organização, segurança e controle de SLA.

Simplificando, SOA promete o que toda empresa sonha: Resultados com tempo de desenvolvimento mais curto e maior aderência ao negócio da companhia.



"Em pesquisa realizada pelo Gartner Group, 4 em cada 5 projetos SOA dentro das empresas falham."

1.2 Promessas SOA

Reusabilidade - Business Services

- O motor que move o SOA é a reusabilidade de componentes de negócio. Desenvolvedores e empresas podem fazer uso de aplicações existentes e expor como serviço, promovendo o reúso das novas peças de software.

Interoperabilidade - Arquitetura desacoplada de serviços

- A visão SOA sobre integração entre clientes e serviços objetiva a clara comunicação entre as partes de software, não importando em que plataforma residem ou foram escritas.

Escalabilidade e flexibilidade

- Os serviços em SOA são desacoplados, portanto aplicações que fazem uso destes tendem a escalar mais facilmente que as aplicações que possuem um acoplamento mais forte.

Falso SOA

Soa como um paradigma de abstração, que tradicionalmente representa uma estrutura base para arquitetura distribuída sem referências de implementação.

Um dos maiores problemas das companhias está na dificuldade de mensurar a extensão de orientação por serviços vs. automação de soluções. Isso não está claro e as organizações costumam usar de maneira errônea.

1.3 Same Old Architecture



Evolução no conceito de Serviços

Technologias usando conceitos SOA :

- Corba / RMI / DCOM / EAI Solutions ...

Década	Unidade de reuso	Escopo
1960	Funções	Programas
1980	Objetos	Packages
1990	Componentes	LOB Applications
2000	Serviços	Companhias

Service Oriented Architecture – é um termo que representa um modelo em que a automação da lógica é decomposta em pequenas e distintas unidades de lógica. Coletivamente, essas unidades formam uma grande peça de automação de negócios. Individualmente, essas unidades podem ser distribuídas – Thomas Erl.

Muitas das ideias por trás do SOA foram implementadas por tecnologias como CORBA, RMI, entre outras.

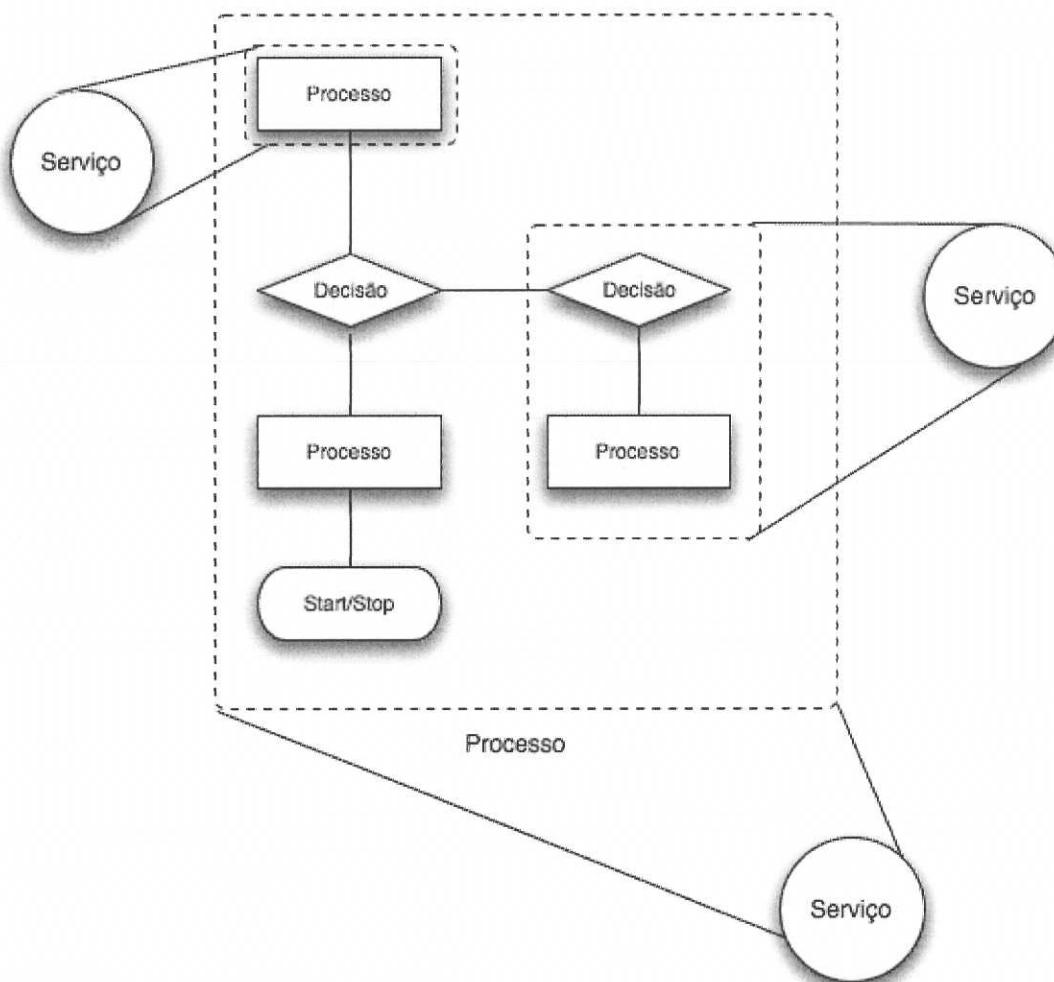
A criação de *remote procedure calls* ajudou no reúso de funções de negócio, promovidas pelo paradigma *Interface Driven Design*. Muitas aplicações corporativas sofreram o “downdsize” do mainframe e foram movidas para plataformas que suportam os paradigmas de computação distribuída.

Durante a década de 90, a internet oferecia estrutura para serviço provisionando o compartilhamento de forma remota através de protocolos como ebXML e SOAP.

1.4 Como os serviços encapsulam a lógica

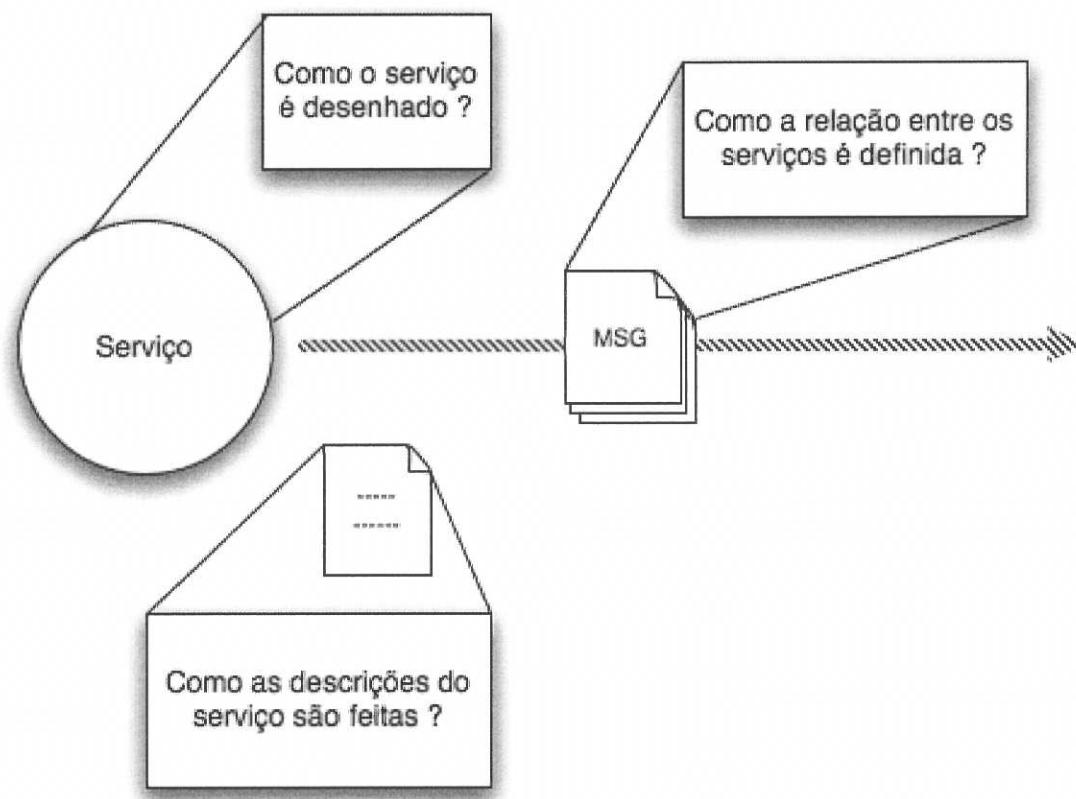
Para obter certa independência, serviços encapsulam lógica dentro de um contexto distinto. Esse contexto pode ser uma tarefa de negócios, uma entidade, ou até um grupo de atividades.

Os interesses endereçados por um serviço podem ser pequenos ou grandes, portanto, o tamanho e escopo de uma representação pode variar. Contudo, serviços podem encapsular ainda outros serviços, neste caso, um ou mais serviços são “compostos”.



1.5 Como os serviços são desenhados

Assim como a orientação a objetos, orientação a serviços tornou-se uma abordagem que introduz seus próprios princípios, definidos em componentes arquiteturais.



Questões abordadas pelos princípios do design orientado a serviços.

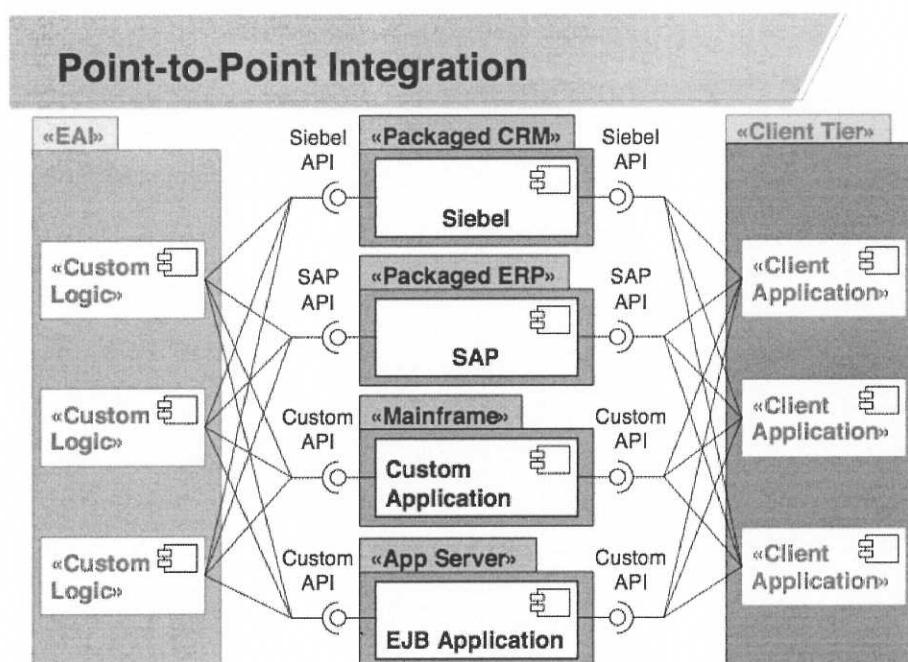
2. Desafios do mundo corporativo

Desenvolvimento de aplicações e problemas de integração:

- Falta de flexibilidade
- Falta de padrões
- Projetos com custos altos e de longa duração

Metodologias tradicionais:

- Point-to-point
- Ferramentas Integradoras (Tuxedo, MQSeries)/middleware/EAI
- Business process-based integration



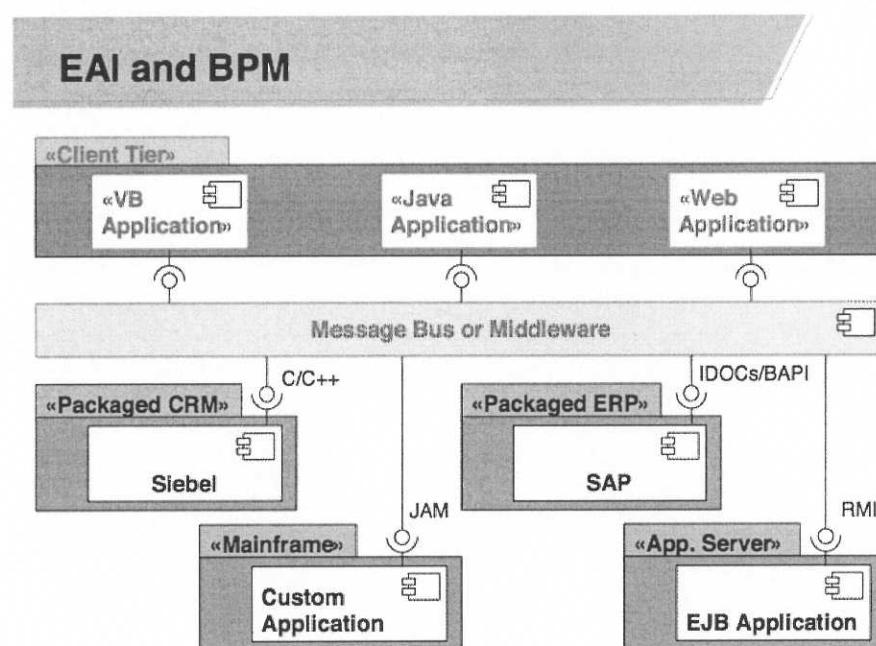
Em uma integração point-to-point aplicações são integradas umas às outras diretamente usando APIs proprietárias, links de integração customizados moldando um forte acoplamento de implementação.

Nesse tipo de projeto, surge um problema de conjunto de habilidades da equipe - skill set, fazendo o mesmo se arrastar por meses ou anos.

Como a arquitetura de integração *point-to-point* era complexa, cara e difícil de manter, outras abordagens foram introduzidas. Uma das mais referidas – EAI – Enterprise Application Integration, a qual se baseava num *message broker* ou um *middleware*. Nesse caso, a conectividade entre a aplicação e o serviço de mensagens era desenvolvido usando APIs proprietárias da ferramenta de integração.

Infelizmente, essa abordagem requeria uso de *APIs* customizadas ou proprietárias de integração e diferentes formatos em cada ponto de integração da solução. Em adição, cada aplicação era fortemente acoplada ao message bus e todas as aplicações necessitavam saber como as outras trabalhavam.

A ausência de padrões de arquitetura e custos limitou bastante os projetos EAI e a indústria começou a olhar para outras soluções de integração que estavam surgindo.



2.1 Princípios chave da orientação por serviços:

Loose Coupling	Serviços mantêm dependências minimizadas, requerendo somente a consciência que um depende do outro.
Service Contract	Serviços são aderentes a um acordo de comunicação coletivamente por uma ou mais descrições e seus respectivos documentos (mensagens).
Autonomy	Serviços possuem controle sobre a lógica que encapsulam.
Abstraction	Serviços ocultam a lógica para o mundo externo
Reusability	A lógica de negócio é dividida em serviços com a intenção de promover o reuso
Composability	Coleções de serviços podem ser coordenadas para formar um novo serviço “composto”.
Statelessness	Serviços minimizam a retenção de informação específicas à uma atividade.
Discoverability	Serviços são desenhados para serem auto-descritivos e serem encontrados e acessados pelos mecanismos de descoberta.

2.2 SOA como arquitetura distribuída:

Nos últimos anos o setor de tecnologia da informação das empresas recebeu pesados investimentos para disponibilizar seus negócios pela internet aos clientes. Durante o fim da década de noventa esse foi um dos focos das empresas para reduzir os custos e manter o crescimento (Carter, 2007). Um exemplo deste fenômeno são as instituições financeiras, que passaram a estimular seus clientes a utilizarem sistemas pela internet e reduziram a quantidade de funcionários para o atendimento ao público.

Atualmente, uma das frentes da computação para auxiliar as empresas a manter o crescimento é por meio da possibilidade de decomposição dos sistemas e a sua integração de diversas formas, desenvolvendo softwares diferentes do paradigma de grandes softwares isolados em seus vários setores. A decomposição e a interligação possibilitam que os setores utilizem partes de softwares de outros setores sem precisar reescrever e duplicar programas. Além disso, as informações podem ser acessadas de uma única forma por todos através de um único serviço que as acesse. “A flexibilidade é o principal combustível para o crescimento dos negócios no cenário atual e ela é proporcionada no setor de tecnologia da informação com a decomposição e interligação dos sistemas.” – Carter.

A característica de execução remota de serviços faz com que possa ser dado ao usuário um aspecto de transparência da aplicação, podendo ele mesmo acessar serviços que estejam em diferentes computadores de forma imperceptível. O acesso aos serviços é dado por uma URI (Universal Resource Identifier) que, por exemplo, pode utilizar um sistema de DNS para fornecer aos usuários diferentes endereços de acesso (Brittain; Darwin, 2003). Baseado em uma mesma URI, cada vez que é feito um acesso a um serviço diferentes computadores podem recebê-lo e gerar uma resposta.

Os web services, segundo Van Steen Tanenbaum (2002), classificam-se como sistemas distribuídos. Eles podem aparentar para o usuário que são partes de apenas um sistema hospedado em um único computador, mas podem estar em diversos computadores independentes.

A área de computação distribuída já possui estudos e propostas para diversos problemas que surgem em sistemas como web services. Uma das situações que podem estar presentes nos web services é a alta demanda gerada por servidores que experimentam diariamente um alto número de requisições dos usuários. Sistemas distribuídos apresentam soluções eficientes para altas demandas, seja através do uso de processamento massivamente paralelo, sistemas de estação de trabalho/servidor, clusters, entre outros.

Um outro importante aspecto estudado em sistemas distribuídos é o escalonamento de processos, ou seja, como os processos executados em um sistema distribuído devem ser alocados para cada recurso do sistema. A distribuição de requisições, de maneira análoga ao escalonamento de processos, também trata como escalar requisicoes para diferentes recursos de maneira eficiente.

Estudar maneiras de realizar a distribuição de requisições é importante para que web services sejam executados de maneira eficiente em clusters. Assim, eles podem fornecer o devido suporte para que SOA apresente o suporte vital no atual crescimento das empresas.

A popularização de SOA e dos web services faz com que as empresas passem a gerar mais demanda sobre esse tipo de aplicação devido ao seu uso intensivo. Para suportar a alta demanda gerada, devem ser aplicados recursos computacionais de alto desempenho, os quais podem ser disponibilizados de maneira centralizada ou distribuída. Na primeira hipótese, é necessário o uso de servidores de grande porte que em geral têm alto custo e grande confiabilidade. A segunda maneira pode ser implementada por um cluster de estações de trabalho, apresentando um custo relativamente baixo e alta escalabilidade.

Com a competitividade do mercado atual, poupar gastos leva ao uso de soluções de baixo custo. Para o uso de clusters, é necessário que seja feita a distribuição das requisições aos servidores de web services de alguma maneira. A distribuição eficiente, sob diferentes pontos de vista, é o fator motivante deste trabalho. Baseado na motivação, o objetivo do trabalho é propor uma arquitetura de distribuição de requisições para softwares provedores de web services, a qual apresente as características de ser flexível, dinâmica e transparente.

- Flexível para permitir o uso de diferentes políticas de distribuição.
- Dinâmico para que se adapte e tome decisões em tempo de execução sobre a maneira como será realizada a distribuição, baseado também no estado atual dos elementos que compõem a estrutura de atendimento.
- Transparente porque a distribuição não deve ser perceptível às aplicações clientes.

2.3 SOA sob ótica de negócios

SOA pode ser analisado por dois pontos de vista. No primeiro analisa-se o contexto da tecnologia, observando todas as possibilidades de aplicações de ferramentas existentes atualmente e no passado, para dividir as aplicações e seus modelos em serviços, disponibilizando-os para acesso de alguma forma.

Outro contexto em que SOA pode ser analisada é sob visão de administradores e projetistas de software. Neste último contexto, SOA é uma abordagem para a tecnologia de informações voltadas para os negócios, transformando um processo que deve ser realizado pelos administradores ou funcionários de diversos níveis em uma composição de diversas tarefas menores.

"SOA é uma abordagem arquitetural de TI dirigida a negócio que suporta a integração de sistemas como tarefas de negócio ou serviços conectáveis e repetitivos. SOA ajuda hoje as empresas a inovar graças a garantia de que os sistemas de TI podem se adaptar rápido, fácil e economicamente para permitir rápidas mudanças nas necessidades da empresa. Ela aumenta a flexibilidade dos processos de negócio, fortalecendo a infraestrutura presente e reusando os investimentos já feitos na área de TI, criando conexões entre aplicações e fontes de informação que apresentam muita diferença entre si." - Thomas Erl

No mundo dos negócios, o uso de serviços pode auxiliar os departamentos de tecnologia de informação a se adaptarem de maneira rápida, fácil e diminuindo os custos com mudanças no decorrer do tempo. Além destes fatores, com o amparo do uso de serviços, as empresas podem aumentar a flexibilidade dos seus processos de negócio, já que podem unir pequenos pedaços de aplicações para gerar outras mais personalizadas para cada departamento, sem a necessidade de desenvolver uma aplicação totalmente nova.

Com o auxílio de algumas ferramentas criadas para este fim, a construção do novo software pode ser simplificada demandando menor tempo para o desenvolvimento.

Serviços menores, que realizam operações mais pontuais trazem as empresas para o mundo de desenvolvimento usando **componentes**. Isso torna o setor de tecnologia de informação mais produtivo e reforça cada vez mais a infraestrutura responsável por fornecer os dados para as aplicações. A forma como os dados são coletados dá-se de uma maneira padrão para todas as aplicações, evitando a criação de novos métodos que podem trazer restrições e possíveis falhas.

Sob o ponto de vista da empresa um serviço pode ser, por exemplo, uma tarefa exercida diversas vezes por um funcionário como a consulta de crédito de um cliente. Ele pode também ser parte de mais de um processo que o funcionário executa. Não se deve olhar para este serviço como um programa que realizará uma consulta em uma base de dados ou entrará em contato com outra empresa de crédito através de alguma tecnologia de comunicação. **Para enxergar serviços em uma empresa deve-se olhar para o seu dia-a-dia e verificar as tarefas de negócio repetitivas que os funcionários fazem, nos passos que são necessários para realizar um negócio.**

As relações que ocorrem entre empresas têm seguido a tendência de se tornarem cada vez mais complexas com o passar do tempo. Fatores como fusões e parcerias, aumento da competitividade, sistemas de produção just-in-time e terceirização de serviços resultam no aumento das aplicações existentes.

Como exemplos de fusões e parcerias podem ser consideradas grandes instituições financeiras e fábricas de bebidas internacionais que se uniram nos últimos anos. Cada uma das empresas que participou das fusões já possuía uma estrutura de tecnologia de informação (aplicações, redes, servidores) bem estabelecida e funcional, que em alguns casos não poderia se comunicar com outras empresas diretamente, devido a barreiras tecnológicas. Em tais situações fica difícil escolher qual o sistema que permanecerá e qual será extinto tendo seus dados migrados. Outra possibilidade é a perda de dados durante a migração, cujo impacto não é difícil de imaginar no caso de uma instituição financeira.

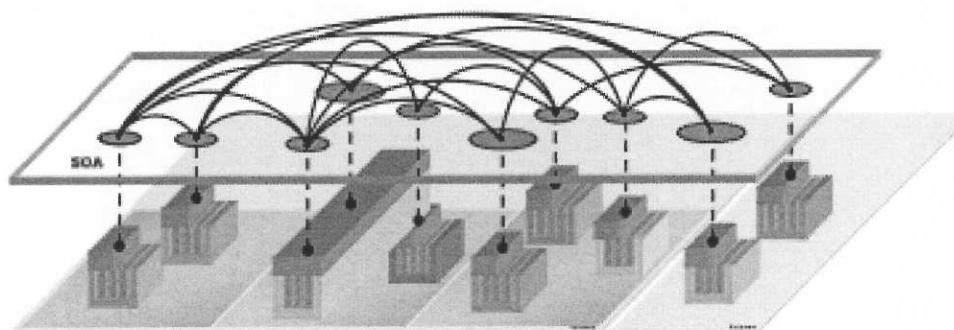
O abandono de um parque tecnológico desenvolvido envolve a desativação de sistemas que podem ter custado milhões de dólares.

O setor de TI dentro de uma empresa tem por objetivo agilizar os negócios por ela realizados. Ele não pode ser o responsável por manter a empresa estagnada por dificuldade integração, obtenção de dados e também não pode consumir uma fatia considerável do faturamento da empresa para o seu funcionamento.

Um recente artigo da IBM prevê que até o ano de 2012 boa parte das empresas exercerá apenas trabalhos especializados, sendo assim, cada empresa terá seu próprio sistema de informações especializado, necessitando de interação business-to-business.

Devido a esses fatores, o mundo dos negócios está apostando muito na SOA. Os números apontam que no ano de 2007 a aplicação de SOA dentro das empresas gerou oportunidade de negócio de sessenta bilhões de dólares, um crescimento de 75% em relação ao ano anterior. É esperado para o ano de 2009 um crescimento nesta área de investimento próximo a 54%, alcançando 143 bilhões de dólares.

2.4 SOA Services



Serviços são componentes de software com funções distintas encapsulados num conceito e consiste de algumas partes:

Um *Service Contract* provisona a abstração e independência de tecnologia entre *service consumer* e *service producer*. Este especifica de forma detalhada a semântica da funcionalidade e parâmetros necessários.

A Interface (WSDL ou IDL) expõe a funcionalidade do serviço para os clientes, que estão ligados através de implementações físicas baseadas em *Stubs* ou *IDL Stubs – Meta Data Driven Contract*.

A implementação do serviço acessa a lógica de negócios que, dependendo da granularidade, pode ser um conjunto complexo de sistemas ou uma simples função de logging.

Alguns serviços possuem relacionamento direto com a lógica de negócios. Aplicações legadas em soluções de middleware normalmente possuem características que podem ser mapeadas pelas interfaces. Substituindo a visão de funcionalidade por interface, podemos atribuir uma visão SOA.

Data-centric services são uma importante parte quando as empresas tentam migrar de processamento batch para algo online ou mais real-time.

3.0 WebServices

O uso de uma arquitetura orientada a serviços implica no particionamento das aplicações que antes eram responsáveis por fornecer dados, por exemplo, para todo um departamento em aplicações menores com funções limitadas. De alguma forma as aplicações menores resultantes desta decomposição devem comunicar-se entre si para gerar o mesmo resultado obtido pela aplicação única.

Esse conceito de divisão e acesso de aplicações através de um protocolo de comunicação caminha ao encontro dos conceitos há tempo difundidos na área da computação distribuída.

Uma gama de protocolos de programação distribuída existem para conectar aplicações por meio de LANs e/ou WANs. Três protocolos comumente utilizados são: CORBA (Common Object Request Broker Architecture), DCOM (Distributed Component Object Model) e RMI (Remote Method Invocation).

Cada um desses protocolos apresenta pelo menos um dos seguintes problemas: incompatibilidade com outros protocolos, dependência de linguagem ou de sistema operacional, modelo de objeto e de passagem de mensagem dependente de fabricante (**Brooks**, 2002). Além desses problemas, há algumas barreiras que limitam o seu uso apenas a ambientes com maior acoplamento. Isso dificulta acessos remotos que necessitem passar por diversas redes ou ainda que possam ser utilizados em uma grande variedade de dispositivos.

A web possibilita que, com uma quantidade pequena de recursos, um dispositivo possa acessar de qualquer parte do mundo um conteúdo nela disponibilizado. No seu princípio a web era uma grande fornecedora de conteúdos estáticos, não havendo formas complexas de interação entre os usuários e os servidores: os usuários apenas requisitavam os dados e dificilmente enviavam dados para o servidor. Com o passar do tempo e o surgimento de tecnologias como o CGI (Common Gateway Interface) tornou-se mais popular a criação de páginas de internet que possibilitavam uma melhor personalização da interação dos usuários com o conteúdo fornecido.

O que tornou isso possível foi a possibilidade dos servidores coletarem dados dos usuários por meio de CGI (*Common Gateway Interface*) e enviarem os dados para linguagens mais elaboradas dentro dos servidores, os quais podem fazer o processamento de diversas maneiras. Desse ponto em diante, passou-se a observar o grande crescimento dos tipos de aplicações que executavam na web sendo o limite muitas vezes apenas a criatividade dos programadores.

O elemento básico para que o usuário interaja com a web é o navegador, que possui a capacidade de interpretar pelo menos uma linguagem simples (HTML) vinda das mensagens do servidor e exibir elementos gráficos padrão para o usuário.

Apesar de haver um padrão na linguagem usada na web (HTML 4.0 ou nos dias atuais o XHTML 1.1) (W3C, 2008), os elementos exibidos para os usuários costumam variar entre os navegadores. Além disso, tornou-se popular o uso de scripts para tornar a interface do usuário mais amigável com as páginas, causando incompatibilidade na maneira como páginas web são exibidas. Além dos fatores envolvidos no lado do usuário, entre os servidores não há uma padronização que oriente como as informações são captadas, ocorrendo problemas de codificação de dados e uma grande variedade de formas de aquisição de informações. Por exemplo, duas páginas que vendem determinado produto podem receber o pedido do usuário de maneira diferente.

A integração de aplicações, que antes encontrava empecilhos na acessibilidade remota, tem na web uma grande aliada. Com a comunicação em ambos os sentidos podendo ser feita sobre o protocolo HTTP, sobre o qual a web é baseada é possível que as aplicações troquem os dados necessários para a sua interação, tal como os usuários fazem em seus navegadores. A união da web e da integração de aplicações permite uma forma de acessos mais simples e com maior alcançabilidade que as outras maneiras citadas anteriormente (CORBA, DCOM e RMI).

O padrão **XML** (*Extensible Markup Language*) foi criado para sanar os problemas com interoperabilidade, portabilidade e padronização. A aplicação do padrão XML com os servidores presentes na web facilitou uma interação mais robusta entre as aplicações e os servidores. Ela possibilitou não só o uso de mensagens simples entre cliente e servidor, mas também a disponibilização de serviços mais complexos por parte dos últimos, com transferência de informação mais complexa e em maior quantidade.

Com **XML** os dados podem ser codificados com tipos de dados padrão, colocados em um arquivo **XML** e enviados para o servidor diminuindo os problemas de compatibilidade.

A exposição na web de servidores que disponibilizam parte de uma aplicação decomposta, que no contexto de SOA são os serviços, caracteriza o início dos web services.

Os web services são muito mais que apenas um servidor que aceita uma requisição para uma aplicação pela web. Eles são constituídos principalmente pelos padrões que tornaram a interação entre aplicações pela web possível. O padrão que fez emergir toda a capacidade dos web services foi o **XML**.

Na essência de sua criação, web services são o elo de integração entre computadores pessoais, outros tipos de dispositivos, bancos de dados e redes, formando uma ligação na qual os usuários podem interagir.

Os web services executam em servidores de aplicações para web e movem as funções dos computadores pessoais para a rede, tornando acessíveis as aplicações a partir de qualquer dispositivo conectado com a mesma, incluindo dispositivos móveis. Sob esse ponto de vista a rede passa a ser uma plataforma de computação e não só um meio pelo qual o usuário tem acesso a conteúdo - **Vaughan-Nichols**

Alguns especialistas dizem que os web services são basicamente chamadas remotas de procedimento com um fraco acoplamento que surgiram como uma alternativa para os métodos com forte acoplamento que requerem aplicações e protocolos específicos.

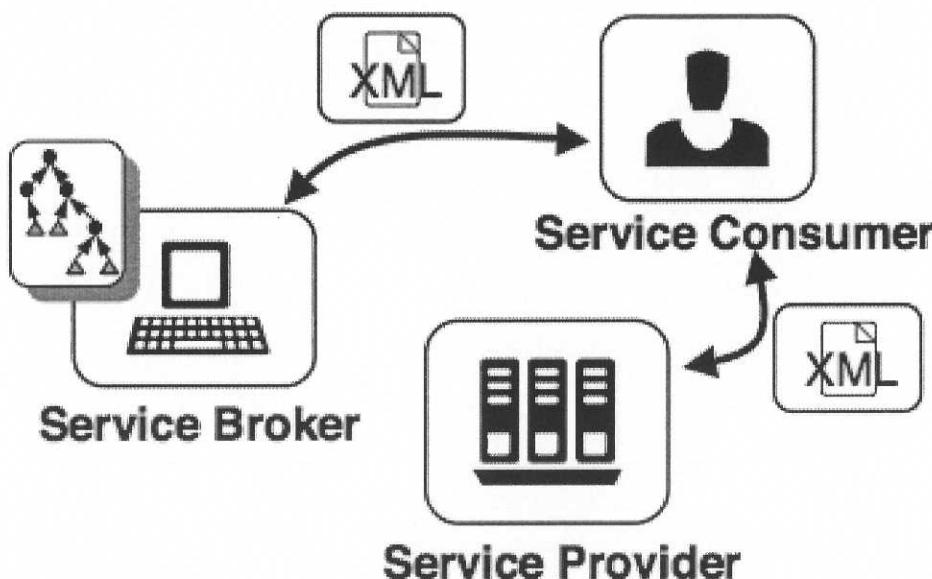
A alma dos web services são os padrões abertos, que possibilitam que qualquer um desenvolva soluções sem a necessidade de licenças de produtos. A presença de padrões abertos é o que difere os web services de outros padrões como o IIOP do CORBA. Alguns dos padrões sobre os quais os web services são construídos são o **SOAP** (*Simple Object Access Protocol*), que é o protocolo utilizado para o transporte de dados das requisições e respostas; e o **WSDL** (*Web Service Description Language*), que é a linguagem pela qual um web service é descrito.

Em relação ao uso do protocolo **SOAP** em web services, ele não é a única maneira sobre a qual estes podem ser construídos. Uma outra abordagem que pode ser seguida é o uso de **REST**. Apesar de **REST** e **SOAP** não serem opostos, pois a primeira é um estilo arquitetural e o segundo um protocolo que pode ser usado em diversas arquiteturas, ainda existem muitos debates sobre esse assunto entre os defensores de cada uma das abordagens.

Um dos problemas existentes no modelo anterior de computação distribuída é a compatibilidade com a plataforma do player.

WebServices oferecem um meio diferente para realizar operações distribuídas, agregando a promessa de uma excelente interoperabilidade.

Papéis do WebService

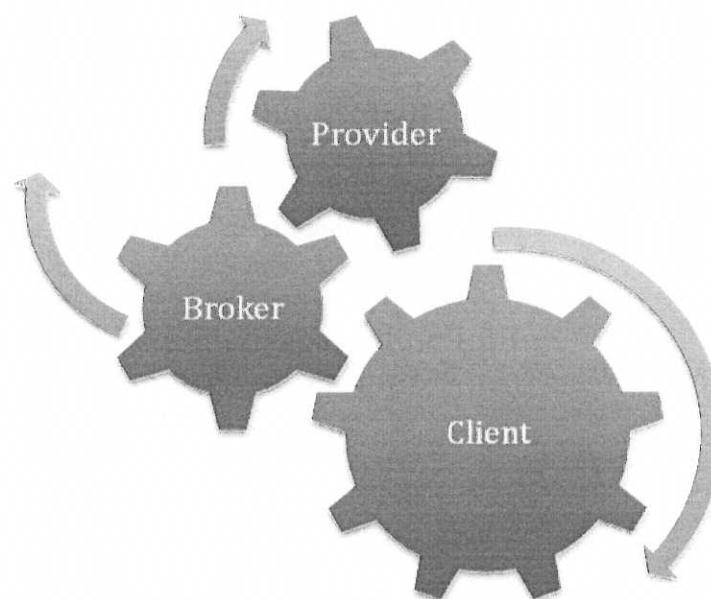


WebServices são representados por três papéis principais:

- **Service Consumer** - usa um WebService.
- **Service Provider** - disponibiliza um WebService.
- **Service Broker** - armazena informação sobre o WebService.

Exemplificando um fluxo de operações:

- 1- O cliente contata o **broker** para procurar um serviço (WebService) que complete a execução do seu negócio.
- 2 - O **broker** identifica o serviço esperado através de uma busca por critérios e retorna ao **cliente** uma descrição apropriada do serviço.
- 3- Usando a descrição recebida do **broker**, o cliente constrói uma requisição de mensagem para a operação de negócio apropriada e envia ao **provider** usando a localização especificada.
- 4 – O **provider** recebe a mensagem do **cliente** e realiza a operação de negócio apropriada e, se requerido, constrói uma mensagem de resposta para ser enviada ao **cliente**.



Padrões WebService

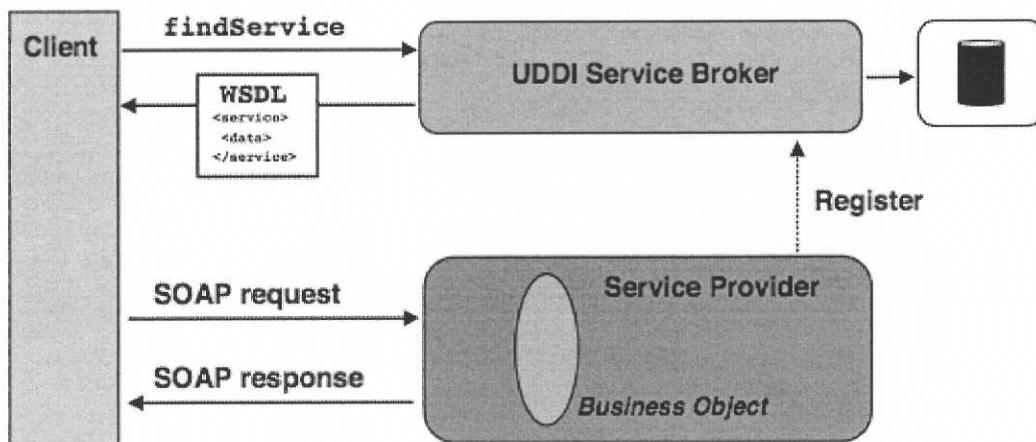
- **Simple Object Access Protocol (SOAP)** descreve o formato de comunicação da mensagem entre as partes envolvidas num Web Service.
- **Web Services Description Language (WSDL)** define um mecanismo para descrever as operações de um Web Service de uma forma neutra às plataformas.
- **Universal Description, Discovery and Integration (UDDI)**, facilita o registro e busca dos Web Services transformando as descrições em hierarquia, facilitando assim a busca.

SOAP é um formato de mensagem baseada em XML, padrão aberto definido pelo W3C. Este mecanismo é utilizado para invocar operações e realizar a “troca de documentos” entre aplicações, como um protocolo de envelope, que pode ser inserido em outros protocolos.

O WSDL também usa um formato baseado em XML, mas é usado para descrever um WebService. Esse documento inclui a descrição do **endpoint**, **location**, **protocolbinding**, **operations**, **parameters** e **data types** – Todos aspectos de um Web Service.

UDDI é um conjunto de especificações que define um formato padrão de registro para o WebService, como o protocolo de mensagens usado para acessar o **registry**. Aqui também são identificadas e classificadas as taxonomias para organização dos Web Services.

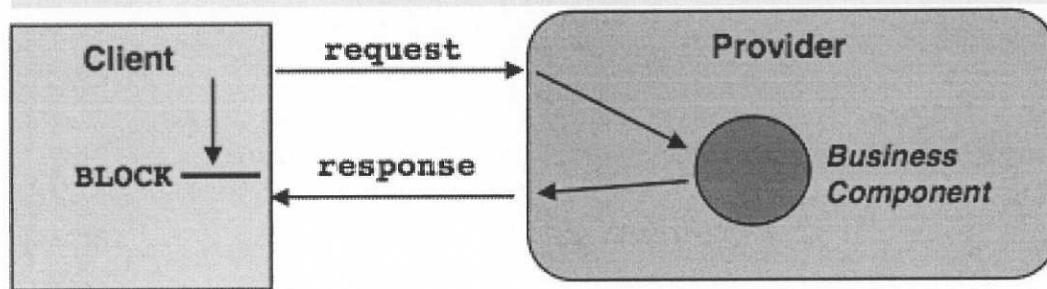
3.1 WebServices funcionando



A seguir o cenário mais comum de um Web Service:

- 1 – O cliente possui uma necessidade de negócio, por exemplo: para computar mensalmente o pagamento de um financiamento.
- 2- O cliente contata o *Broker* ou *Registry* usando *UDDI* para procurar um *provider*, a fim de realizar a requisição.
- 3- O *broker* retorna um WSDL anteriormente registrado para o FinanciamentoAutomotivo, o qual oferece o serviço.
- 4- O cliente usa o WSDL para construir uma requisição SOAP incluindo os parâmetros necessários para a operação(Cliente, Carro, Taxa, entre outros).
- 5 – O *service provider*, FinanciamentoAutomotivo, processa a requisição e retorna uma resposta para o cliente, indicando a data para o pagamento.

3.2 Operações Síncronas:

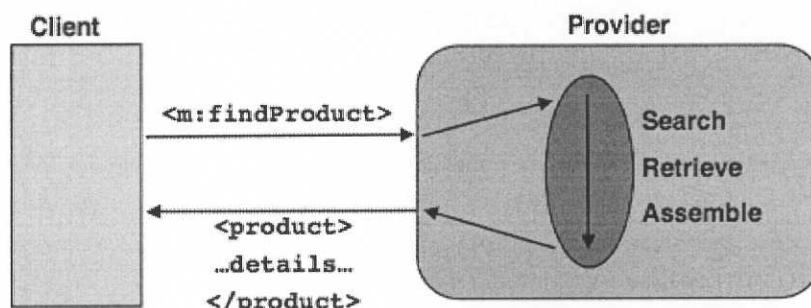


- Lembra bastante a forma de invocação RPC
- Fortemente acoplada entre o cliente e a camada de negócios
- São ideais para operações de negócios rápidas com retorno de resultados
- São melhor usadas pelas aplicações da própria companhia.

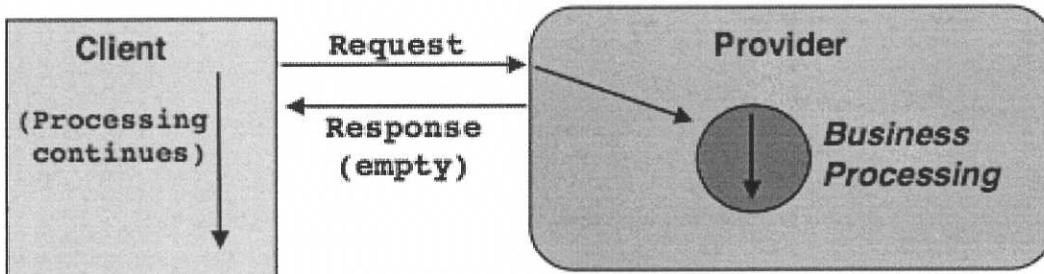
Um Web Service Síncrono é um dos mais comuns encontrados, e lembra bastante a maneira tradicional RPC – *Remote Procedure Call*, ou chamadas à computação distribuída usando RMI – *Remote Method Invocation*.

A resposta é requerida, e o cliente da aplicação não pode continuar seu processo enquanto não recebe a resposta.

Infelizmente, invocando um Web Service síncrono sua operação fica fortemente acoplada – *client* e *provider*. Este tipo de design deve ser utilizado somente em operações rápidas, onde o ambiente é controlado e a comunicação e o retorno dos resultados são simples, para que o *service provider* possa garantir o processamento de resposta.



3.3 Operações Assíncronas:



- Retorna uma resposta SOAP vazia se a resposta for requerida.
- Desacopla o cliente do processamento de negócio
- Promove o desacoplamento e melhor uso dos recursos
- É ideal para integração com parceiros externos à companhia

Web Service Assíncrono não requer que o *client* fique esperando pelo processamento das operações. Geralmente (dependendo do protocolo de transporte) uma resposta é retornada ao cliente, mas essa não contém nenhuma estrutura de dados para o mesmo utilizar. É apenas uma resposta para identificar que o processo de negócio foi invocado.

Se uma resposta é requerida, ao final do processamento ("fire and forget" – similar ao envio de um e-mail o qual não necessita de resposta), então o *provider* pode notificar o cliente sobre o término do mesmo. Se isso não for possível (às vezes isso acontece por causa de firewalls que recebem solicitações não requisitadas), o *provider* armazena ou responde ao *poll* de *client*.

Com um Web Service assíncrono, tanto o *client* quanto o *provider* podem continuar a processar de forma independente, o que promove uma melhor performance. Para cada instância, requisição SOAP, uma mensagem é postada na fila do servidor e processada de maneira única, mantendo o servidor em pé mesmo com um alto volume de requisições. Além disso, o cliente não precisa esperar durante um grande período enquanto o processamento é realizado.

3.4 Exercício, construção de WebServices

EXERCÍCIO – Executar 2 serviços construídos em cima de WebServices.

1. O primeiro serviço será síncrono, ou seja, retorna algo ao requisitante da operação.
2. O segundo serviço será assíncrono, disparando o processamento e liberando o cliente do processamento.

3.5 SOAP – Simple Access Protocol

Um dos desafios quando se utiliza a computação distribuída é garantir a interoperabilidade de sistemas. Nos últimos anos, o surgimento de um protocolo chamado SOAP (Simple Object Access Protocol) apresentou-se como uma boa arma para atacar o problema de interoperabilidade.

O protocolo SOAP é relativamente leve, baseado em XML, projetado para que aplicações possam trocar informações e realizar pedidos de execução de procedimentos em aplicações remotas.

SOAP é constituído por algumas partes:

- A primeira é o envelope, que engloba todo o conteúdo da mensagem e diz ao receptor como processar os dados contidos na mensagem através de um documento de descrição.
- A segunda garante a extensibilidade do SOAP, podendo nela ser configurados alguns parâmetros sobre o tráfego da mensagem e como ela deve ser interpretada por onde passa.

- A terceira parte de uma mensagem SOAP descreve como as informações enviadas devem ser usadas, ou seja, qual o procedimento deve ser disparado, os tipos de dados passados e os tipos de retorno para o procedimento. Também são enviadas algumas regras de como o protocolo HTTP será utilizado (no caso do uso de HTTP). Além do uso de HTTP, SOAP pode ser transportado por outros protocolos como SMTP (Simple Mail Transfer Protocol), FTP (File Transfer Protocol), JMS (Java Message Service).

O protocolo SOAP se diferencia de RMI, CORBA ou DCOM uma vez que ele se concentra no conteúdo, desacoplado da implementação e das camadas de transporte. Ele não existe por si só além da abstração da especificação.

A figura SOAP 1 apresenta a estrutura básica de uma mensagem SOAP. Nela é possível ver as três partes que compõe a mensagem: o envelope, o cabeçalho e o corpo.

```
<Envelope xmlns ="http://schemas.xmlsoap.org/soap/envelope/">
  <Header>
    ...
  </Header>
  <Body>
    ...
  </Body>
</Envelope>
```

SOAP 1 fig

A figura SOAP 2 mostra o cabeçalho SOAP contendo um parâmetro chamado mustUnderstand, que nesse caso possui o valor 1, dizendo que por todos os lugares onde a mensagem passar, todos os campos contidos no cabeçalho devem ser compreendidos. Além desse tipo de informação, no cabeçalho podem ser adicionadas informações sobre as extensões que podem ser utilizadas nos web services como a **WS-Addressing**, **WS-ReliableMessaging** entre outras.

```
<Header>
  <x:CorrelationID
    xmlns:x="http://www.xmltc.com/tls/headersample/"
    mustUnderstand="1">
    0131858580-JDJ903KD
  </x:CorrelationID>
  <wsrm:Sequence>
    <wsu:Identifier>
      http://www.xmltc.com/railco/seq22231
    </wsu:Identifier>
    <wsrm:MessageNumber>
      12
    </wsrm:MessageNumber>
    <wsrm:LastMessage/>
  </wsrm:Sequence>
</Header>
```

SOAP 2 fig.

Na figura SOAP 2, os rótulos marcados com **wsrm**, dão informações sobre a extensão **WS-ReliableMessaging** - Thomas Erl, 2005

O conteúdo do corpo pode ser diferente para o acesso a diferentes web services. A **WSDL** é quem define o seu conteúdo, descrevendo como o web service deve ser utilizado, os tipos de dados e os métodos disponíveis. A WSDL do serviço é obtida durante o processo de descoberta realizado pelo cliente. O corpo é um elemento obrigatório dentro do elemento envelope, diferentemente do cabeçalho. Ele carrega toda a informação (*payload*) que realmente representa a interação entre as partes envolvidas na comunicação formatada em XML.

Os namespaces1 do XML são amplamente utilizados para descrever o conteúdo do pacote.

```
<Body>
  <exemplo:objeto xmlns:exemplo="http://www.servidor.br/">
    <exemplo:ID>
      001
    </exemplo:ID>
    <exemplo:Atributo1>
      Valor 1
    </exemplo:Atributo1>
  </exemplo:objeto>
</Body>
```

SOAP 3 fig.

Ainda dentro do corpo de uma mensagem SOAP pode haver mais um elemento que descreve estados de falha na comunicação. A figura SOAP 4 contém um corpo de mensagem SOAP com um elemento de falha que descreve um problema no campo mustUnderstand exemplificado na figura SOAP 3. O elemento de falha apresenta um código de erro no campo **faultcode** que é definido entre muitos outros na especificação do **SOAP**. Os outros campos **faultstring** e **detail** fornecem uma descrição para ser compreendida por seres humanos para amparar a correção de possíveis falhas.

```
<Body>
  <Fault>
    <faultcode>
      MustUnderstand
    </faultcode>
    <faultstring>
      header was not recognized
    </faultstring>
    <detail>
      <x:appMessage
        xmlns:x="http://www.xmltc.com/tls/faults">
        The CorrelationID header was not
        processed by a recipient.
      </x:appMessage>
    </detail>
  </Fault>
</Body>
```

SOAP 4 fig.

3.6 Provedores de WebServices

Os provedores são a estrutura que podem ser utilizadas para amparar o uso de web services, seja no seu desenvolvimento ou publicação. Serão apresentadas algumas dessas que tem mais expressividade em relação ao seu uso na comunidade e suas características principais, destacando pontos fortes e fracos em suas implementações.

Software Infrastructure

Os *web services* são basicamente pequenas aplicações que depois de implementadas precisam ser publicadas na rede para que outras aplicações possam acessá-las.

Para tornar a tarefa de publicar *web services* mais simples, existem softwares que encapsulam todo o tratamento de conexões através de protocolos e direcionam apenas as informações relevantes para a aplicação que implementa o serviço. Se não existissem os softwares provedores, todos os web services deveriam implementar um pequeno servidor para a recepção de requisições. Além disso, seriam necessários métodos para que vários web services pudessem coexistir e diferenciar para qual deles as requisições se destinariam.

Os softwares que agrupam os *web services* e os tornam públicos são os **provedores** de *web services*. Atualmente, existem muitas implementações destes provedores, cada um com suas características particulares, o que torna difícil a escolha de um deles e também eleger o melhor entre todos. Assim como na discussão sobre **SOAP** e **REST**, a escolha entre um provedor deve levar em conta qual é a abordagem desejada na implementação e publicação dos serviços.

Entre os softwares provedores mais conhecidos para *web services* estão o **XFire** (Codehaus, 2008), **Celtix** (Object Web, 2008), **JBoss** (jboss.org, 2008)(Red Hat, 2008a), **WebSphere** (IBM, 2008), **Jetty** (Webtide, 2008), **Apache Tomcat** (The Apache Software Foundation, 2008a), **Apache Axis2** (The Apache Software Foundation, 2008b) e **Oracle Weblogic** (ex. BEA 2008).

* - Essa lista faz menção à plataforma Java – os softwares citados estão em desenvolvimento por mais tempo e possuem grupos de usuários relativamente amplos, cada grupo defendendo o seu software e apontando seus pontos fortes.

A área de SOA está em ascensão, por isso estão surgindo vários outros provedores e adaptações de provedores web que podem fornecer suporte à publicação de *web services*.

Os softwares que fornecem suporte aos web services podem atuar em duas frentes, disponibilizando uma interface de programação para aplicações que auxilia os desenvolvedores a realizarem o envio, recebimento e publicação encapsulando esses detalhes, e também atuando como provedores que expõem os serviços na rede.

Alguns dos softwares realizam ambas as tarefas. Outros apenas fornecem as APIs e utilizam outros softwares para a publicação.

Entre os softwares que apenas expõe os web services estão o **Jetty** e o **Apache Tomcat**. O **WebSphere**, **WebLogic** e o **JBoss** possuem todas as funções para fornecer e implementar *web services*.

Celtix, **XFire** e **Apache Axis2** fornecem uma API para desenvolvimento e necessitam de outros softwares para fazer a publicação dos serviços. Como pode-se perceber, existem diversas combinações que podem ser feitas entre eles além de que, existem outras iniciativas menos conhecidas ou em processo de desenvolvimento para amparar o uso de web services.

O Oracle SOA Suíte é uma plataforma de integração da Oracle. Ele é composto por um conjunto de servidores, serviços e ferramentas necessárias para amparar o uso em larga escala de aplicações web fornecendo interoperabilidade. O seu conceito é garantir que a integração de softwares seja robusta, flexível e confiável.

Sua base é o servidor de aplicação, Weblogic, na especificação JEE (Java Enterprise Edition) sobre o qual todas as outras funcionalidades trabalham.

O **WebSphere** possui extensiva documentação, mas seu uso é relativamente complexo. Isso ocorre porque ele agrupa muitas funcionalidades de diversas tecnologias. Para o fim deste trabalho, o WebSphere não se adequa, pois nas suas versões não abertas não é possível realizar modificações. Outro ponto é o preço elevado da licença.

A versão **Community Edition** é baseada em softwares do grupo Apache, como o **Apache Tomcat** e **Axis2**. Então, uma modificação nestes dois softwares pode refletir-se no **Apache Geronimo**. Outra característica que torna este software inapto para o trabalho é a alta complexidade e a grande quantidade de documentação a ser estudada para colocá-lo em funcionamento e para analisar sua estrutura, tornando-se inviável devido a relação tempo x recurso humano.

O servidor de aplicação **JBoss** surgiu em 1999 e com o seu crescimento e popularização houve a criação do JBoss Group em 2001, para fornecer suporte especializado.

Em 2004, o JBoss Group transformou-se em uma empresa (Fleury et al., 2005). Com apenas sete anos de existência, o JBoss, que sempre foi baseado em software livre, passou a estar entre os líderes no ambiente empresarial como solução para servidores de aplicação. O seu sucesso chamou a atenção da Red Hat, também uma das líderes em soluções empresariais mundial, que adquiriu o JBoss em maio de 2006 por aproximadamente 400 milhões de dólares. O crescente interesse por SOA foi o propulsor de ambos os lados, JBoss e Red Hat (Red Hat, 2008).

Atualmente o **JBoss** continua sendo fornecido sob licença livre seguindo os padrões da Red Hat, apresentando versões gratuitas e versões empresariais. Ambas as versões são iguais, sendo a diferença entre elas o suporte prestado.

Assim como o **Weblogic**, o **JBoss** apresenta uma solução completa para integração de aplicações empresariais atuando sobre a maioria das tecnologias Java voltadas para este fim.

Uma das frentes de atuação do JBoss é em relação a web services, fornecendo todo o ambiente para a publicação dos serviços e a interface de programação de aplicação, atualmente baseada em anotações. O JBoss também possui o Apache Tomcat embarcado podendo haver a execução de web services através deste último, em conjunto com o Apache Axis2 ou a sua própria interface interna.

A documentação do JBoss, assim como do Weblogic e Webphere, também é extensa, fornecendo detalhes sobre todas as suas funcionalidades.

Outros dois softwares que estão relacionados com web services são o **XFire** e **Celtix**.

Ambos não são propriamente provedores de web services em si. Eles são interfaces de programação de aplicação que fornecem um conjunto de bibliotecas e funções para tornar a implementação de web services mais simples.

O software **XFire** apresenta suporte à transferência de mensagens **SOAP** via **HTTP**, **JMS** (Java Message Service) e **XMPP** (Extensible Message and Presence Protocol). Ele é baseado no método de manipulação de arquivos chamado **StAX** (Streaming API for XML) que é baseado em pull, isto é, a aplicação não passa o controle do parsing de dados XML para o parser, mas sim comanda o parser para que ele atue apenas nos elementos XML desejados.

Essa abordagem faz com que não haja necessidade da aplicação manter arquivos XML muito

grandes inteiros na memória, além disso, pode ser usado um parsing retardado, onde o caminhamento pela árvore XML é realizado apenas sob demanda, não carregando o documento inteiro (Sun Microsystems, 2008).

O **XFire** clama por ter desempenho superior à versão 1 do Apache Axis, porém não há comparações com a versão 2. A publicação dos serviços através do XFire é realizada através de qualquer container de servletrs como Apache Tomcat ou Jetty.

O software **Celtix** se baseia em um modelo de **ESB** (Enterprise Service Bus)2 e também fornece uma API para a criação de serviços. Ele foi desenvolvido por uma empresa chamada *Object Web* e apresenta basicamente duas versões, uma gratuita e outra empresarial. A versão gratuita deixou de ser desenvolvida dentro da *Object Web* e passou para o grupo **Apache**. A versão empresarial também saiu das mãos da companhia que o criou e passou para um grupo chamado **Open.IONA** (IONA, 2008) onde o software acabou mudando de nome. A publicação de serviços com o **Celtix** ainda pode ser feita através do Apache Tomcat. Ele também oferece ligações para outras ferramentas como **Spring** (Spring Source, 2008) e **ActiveBPEL** (Active Endpoints, 2008).

Ambos os software mencionados anteriormente (**Celtix e XFire**) passaram por um processo de fusão e passando para o controle do grupo Apache em sua incubadora de projetos. O produto gerado pela fusão possui o nome **CXF** possui uma API bem estabelecida e estável.

Em uma outra categoria de software está o **Jetty**, um servidor web completo escrito inteiramente em Java e que possibilita o uso de extensões para fornecer conteúdo dinâmico.

Ele possui características semelhantes ao **Apache Tomcat** como a configuração através de arquivos XML, a capacidade de ser um container para servlets e a ligação com outros softwares como o **JBoss**, **Geronimo** e **JOnAS** (Ow2, 2008).

Além da ligação com estes softwares, que possuem uma função mais ampla, o Jetty e Apache Tomcat também podem ter inseridos dentro deles APIs para web services como **XFire**, **Celtix** e **Apache Axis**.

Um fator que faz o Apache Tomcat se destacar em relação ao **Jetty** é que o primeiro é a implementação de referência de servlets e, como muitas das APIs para web services que usam como base servlets, o Apache Tomcat oferece maior compatibilidade.

Apache Axis Framework

O Apache Axis ou Axis (Apache eXtensible Interaction System) é um framework para construir clientes, servidores e gateways para **SOAP**. Ele possui implementações voltadas para Java e para C++ e inclui, além do suporte ao desenvolvimento, as seguintes características: um servidor simples para receber as requisições, ligação através de servlets com o Apache Tomcat, suporte atualizado a WSDL, ferramentas para geração de classes baseado em WSDL e ferramentas para monitoração de pacotes.

Considera-se que a versão atual do Axis é a terceira geração de frameworks SOAP. A primeira geração foi marcada pelo SOAP4J, desenvolvido pela IBM por volta de 1996. A segunda geração iniciou-se com a passagem do controle do SOAP4J para o grupo Apache, criando o Apache SOAP, que foi seguido pelo desenvolvimento do Axis.

A motivação para sua criação caracterizou-se pelas diversas propostas existentes para a sua implementação que convergiam para um ponto em comum: a criação de um conjunto de funcionalidades pequenas que poderiam ser compostas para cumprir funções maiores aumentando a sua flexibilidade. O conceito envolvido nas propostas era o de criar uma cadeia de mensagens que pudessem ser repassadas para cada pequena função exercer sua tarefa sobre ela.

Em agosto de 2004 foi introduzida a versão 2 do Axis, ou simplesmente Axis2. Ela apresenta uma nova arquitetura em relação à versão 1 (criada apenas como prova de conceito (Jayasinghe, 2008)) e possui maior flexibilidade, eficiência e possibilidade de configuração em relação à anterior. A última versão suporta não só o uso do protocolo SOAP (versões 1.1 e 1.2), mas também o desenvolvimento de web services baseados em REST. Ela clama por apresentar modularidade e dar melhor suporte para XML, não só o utilizando em sua configuração, mas também nas bibliotecas para o usuário.

As extensões para web services **WS-ReliableMessaging**, **WS-Coordination**, **WS- AtomicTransaction**, **WSSecurity** e **WS-Addressing** possuem implementações através de módulos que podem ser adicionados ao Axis2.

O Axis2 possui um modelo de objetos chamado AXIOM (Axis Object Model) que possibilita um melhor desempenho usando processamento de XML baseado no método pull.

O uso do AXIOM faz com que não seja necessária a criação de stubs para a comunicação com web services. Como o AXIOM trabalha em contato mais direto com XML, sendo seu desempenho superior ao dos stubs.

3.7 Distribuição das requisições

O escalonamento de processos é um tema muito pesquisado na computação distribuída onde, de maneira simplista, vários processadores distribuídos recebem programas para serem executados.

Quando se trata do contexto de SOA, que pode ser implementada por meio de sistemas distribuídos, ao invés de apenas processos, podem ser consideradas requisições a diversos serviços como objeto da divisão.

A analogia aos processos pode ser considerada pelo fato de requisições, no momento em que chegam a um servidor, fazem com que este último execute certo processamento (podendo executar um processo ou uma thread). Por ser um problema comum no dia-a-dia da computação distribuída, o problema de escalonamento possui vasta literatura, surgindo várias nomenclaturas e taxonomias.

Apesar das várias definições e discussões na área, existe um ponto de partida no estudo sobre escalonamento. Casavant e Kuhl diferenciam o escalonamento como local e global. O primeiro é relacionado à situação onde há a atribuição de processos a um único processador, cenário encontrado em um sistema operacional, por exemplo. O segundo é definido como um recurso para a gerência de recursos.

Os elementos para o escalonamento global são os consumidores, o escalonador e os recursos. Os consumidores são os usuários e suas aplicações. Os recursos podem ser processadores, memórias, dispositivos de comunicação, entre outros. O escalonador é quem realiza a alocação dos recursos aos consumidores e é composto por uma política que dita como serão as regras para a alocação.

Com o uso crescente de clusters para atender requisições a web services, faz-se necessário o uso de algum instrumento que faça o escalonamento, nesse caso o escalonamento global, das requisições para cada um dos nós do cluster.

O escalonamento ou distribuição das requisições pode ter diversos fatores motivantes. Se for considerado um cluster onde todos os nós tenham o mesmo poder de processamento (ambiente homogêneo) e que todos eles possam atender a uma certa requisição no mesmo intervalo de tempo, pode ser que o envio de uma requisição para cada nó (*round-robin*) seja uma boa abordagem. Essa solução pode ser implementada de maneira estática sempre com a mesma regra de distribuição.

Nesse mesmo ambiente, pode ser que haja além das requisições, outros processos em execução nos nós, tornando-os inaptos à execução. Para solucionar esse problema, segue-se uma abordagem dinâmica, que reaja ao estado atual do sistema.

A tarefa de otimizar o uso de um sistema distribuído, ou seja, encontrar a melhor solução possível para o atendimento de uma requisição é bastante complexa. Na maioria dos casos ela é um problema NP-completo, podendo ter solução viável apenas em casos específicos como, por exemplo, um número fixo de processos com tempo de processamento conhecido e dois processadores. Assim, torna-se mais interessante realizar a distribuição baseada em heurísticas, com parâmetros que afetam a plataforma de maneira indireta

Nos softwares como o Weblogic a distribuição das requisições é feita de maneira simples, sem conhecimento do estado dos nós do cluster. A política usada neles é round-robin (podendo variar para outras), enviando as requisições a medida em que chegam para cada nó.

Quando já foram atribuídas requisições para todos os nós, a distribuição recomeça pelo que recebeu a primeira requisição e continua da mesma maneira. Um esquema semelhante ao round-robin também é usado nas outras abordagens como distribuição por DNS e por NAT.

Há uma lacuna a preencher no atendimento a web services levando em consideração as informações do próprio serviço, como sua demanda por comunicação, por acesso a unidades de armazenamento ou processamento. Também que realizem a distribuição relacionando o serviço com o estado das máquinas, como a sua carga de processamento atual, tráfego de rede, tráfego em dispositivos de armazenamento, número de processos em execução, entre outros.

Outra característica que torna interessante uma distribuição mais eficiente de requisições, é a flexibilidade referente ao uso de várias políticas ou a criação e incorporação de novas políticas aos softwares.

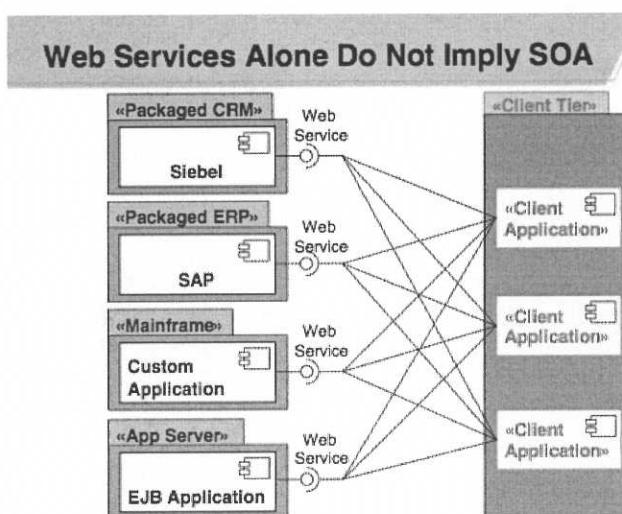
Para que a distribuição seja realizada levando em conta a carga de trabalho dos nós do cluster é necessário que de alguma forma eles sejam monitorados e que essas informações sejam disponibilizadas para o escalonador.

Algumas empresas como RedHat, IBM e Oracle (entre muitas outras), possuem um *stack* completo para ofertar ao mercado meios de preencher a lacunas, como monitoria de carga, desacomplemento, estado de máquina entre outros .

3.8 WebServices sozinhos não implicam em SOA

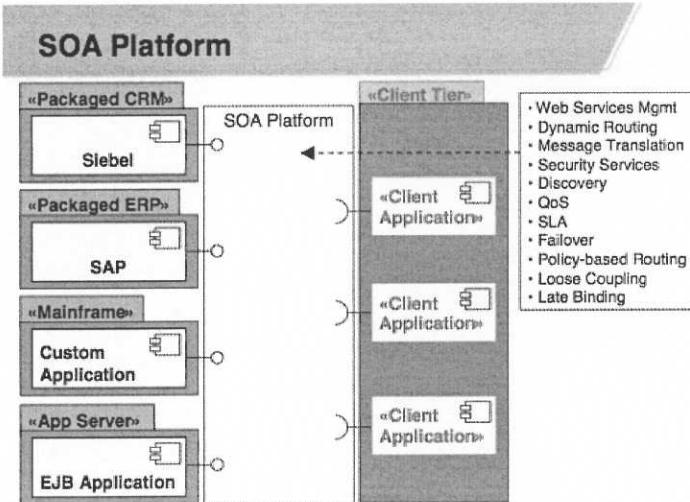
Analizando os problemas do cenário de integração, criar simplesmente uma conexão por intermédio de um webservice não faz a arquitetura ser orientada a serviços – SOA.

O que teríamos como produto final, seria o deploy de uma aplicação point-to-point, com algum benefício de utilizar conexões baseadas em padrões abertos de mercado.



Webservices podem ser implementados usando uma série de plataformas diferentes (DotNet, Java, Ruby, Python...) mas uma plataforma SOA necessita prover uma arquitetura com capacidade de gerenciar uma quantidade elevada de serviços desacoplados baseados na especificação ou tecnologias similares.

A idéia de uma plataforma SOA refere para o compartilhamento de serviços e infraestrutura a qual cria um core de arquitetura, que permite descobrir dinamicamente os serviços e recursos que serão gerenciados. Outras funcionalidades são necessárias para incluir um roteamento de serviços, failover (tolerância à falhas), bem como segurança e políticas de SLA.



4.0 SOA Stack - Infraestrutura

SOA não é inerente a produtos ou tecnologias específicas, mas a fim de aumentar a eficiência e reduzir o tempo de desenvolvimento de infraestrutura, o documento mostra conceitos e ferramental disponível para cumprir tal tarefa. Esse corresponde à:

- Como criar uma cadeia de serviços em um fluxo de processo - WorkFlow e Orquestração.
- Gestão do ativo de serviços - Governança
- Controle de segurança
- Conectividade e desacoplamento

Esse stack leva consideração os principais players de mercado, sobre tudo a Oracle, cuja a qual é apontada pelo Gartner -2009, como a primeira no ranking de soluções SOA.

4.1 Workflow e Orquestração

Uma implementação SOA madura, utiliza uma camada de orquestração de serviços que é enfocada em desenhar o processo de negócio da solução de forma ágil, aplicando agregação de dados (data aggregation), definição de processos (BPM) e workflow criando na ponta os serviços desejados pelo negócio.

Desenhar e desenvolver a Orquestração de serviços, exige conhecimento em integração de processos e regras, camada onde a lógica de negócio é implementada e publicada como um serviço rico – *enriched service*.

Em compromisso com a promessa de agilidade, essencialmente separe os serviços em *Core Business* e *Logic Control*. Se a separação é completa, mudanças nos processos existentes e introdução de novos processos devem acontecer sem problemas, pois o impacto é minimizado, contribuindo para reduzir redundâncias e inconsistências.

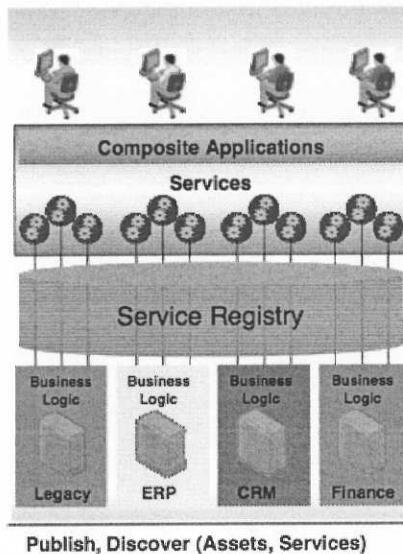
Um sistema de BPM (Business Process Management) oferece uma abordagem de modelagem passo-a-passo para implementar a lógica de negócio. Essa implementação pode cobrir regras de negócio complexas que requerem manutenção do estado da informação, como loops, testes e execuções paralelas. Estes novos serviços, são reutilizáveis na estrutura como componente para construção de outros serviços.

4.2 SOA Management

SOA Management se refere a gestão do conjunto de serviços, políticas afim de estabelecer as melhores práticas possibilitando às equipes de TI das organizações ganhar visibilidade sobre o SOA, dirigido ao reuso dos serviços, definir e reforçar as políticas e gerenciar todo o seu ciclo de vida.

Ferramentas de gestão de Webservices (Web services Management Tools), gerenciam toda a especificação WS*, bem como o portfólio de serviços. Os serviços são registrados e colocados num lugar onde a organização começa a ter um sistemático esquema de gestão dos mesmos.

4.3 SOA Governance - Registry

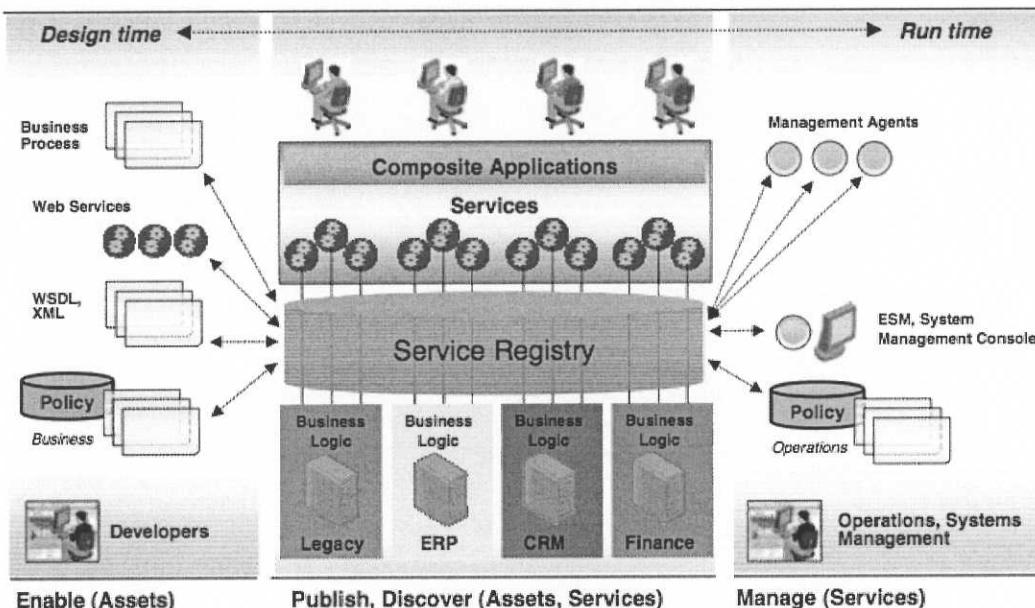


Um serviço de resguardo SOA é uma peça fundamental para a construção da infra-estrutura para nossos serviços.

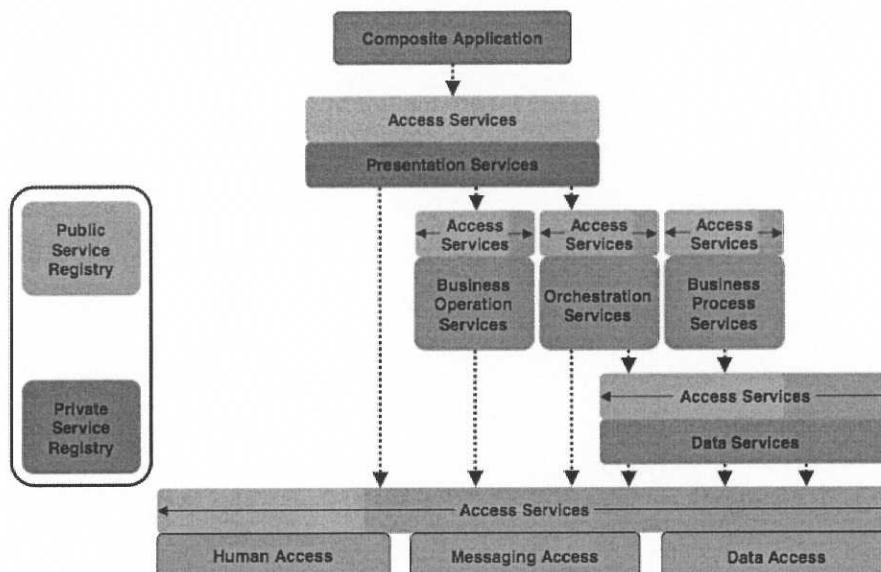
Serviços possuem contratos, associados a documentos XML (modelo canônico) e políticas, como de acesso e segurança, que são publicadas num sistema de registros (Registry) para transceder os silos das organizações e expor para toda a companhia. Esta prática cria visibilidade na hora de criar e descobrir mecanismos, incluído o uso de classificação em tempos de *design* e *run time*.

Como os serviços de negócio (business services) são registrados e promovidos em produção, agentes de gerenciamento (*management agents interface*) juntamente com o *registry* provisionam a gestão desses serviços.

As ferramentas de console dos sistemas de monitoria trabalham de forma bidirecional para garantir que as políticas de run-time são reforçadas e informações de status são enviadas ao registry a fim de refinar a localização dos recursos – discoverability.

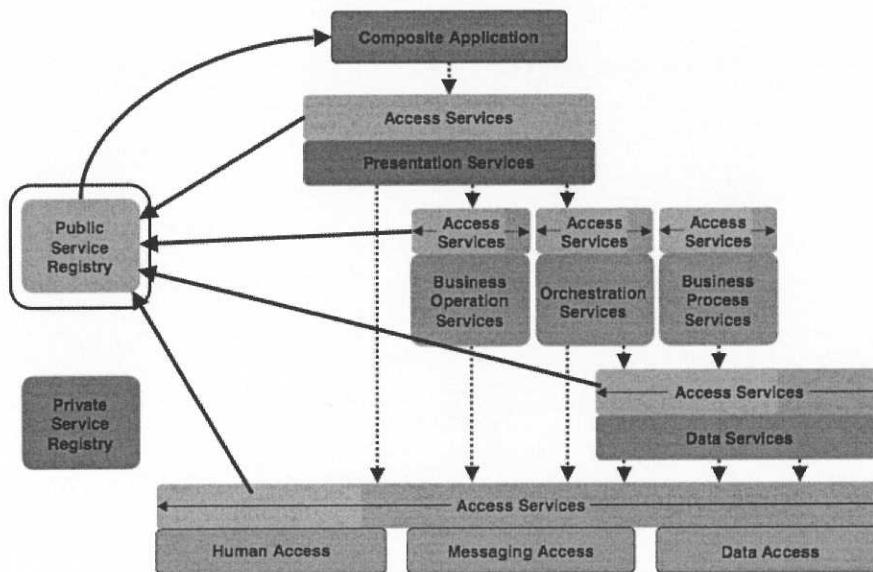


4.3.1 Public & Private Service Registry

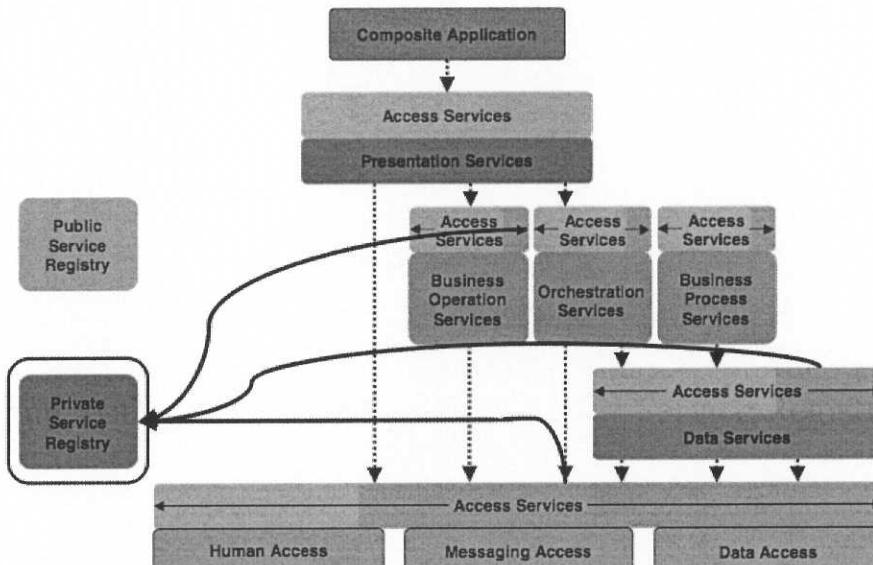


O serviço de registro público (*Public Service Registry*) contém definições dos serviços compartilhados públicos que são disponíveis tanto para aplicações compostas quanto para outros serviços. (*Shared Services*)

O serviço de registro privado contém definições para os serviços privados (*Private Shared Services*) que estão somente disponíveis para outros serviços compartilhados, mas não são expostos externamente (*Public*). Ele possibilita que desenvolvedores descubram e reusem serviços existentes privados.



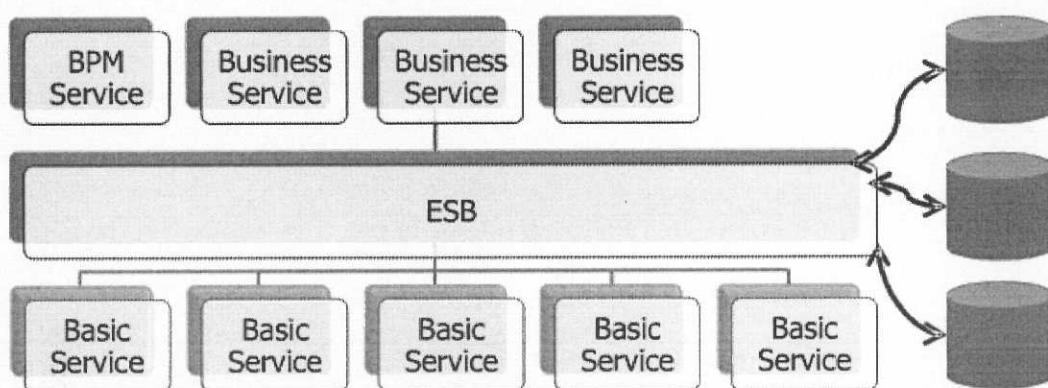
Portlets, business services, data services service provider são expostos como serviços compartilhados público para aplicações compostas e registrados em um Public Service Registry onde poderão ser descobertos pelos desenvolvedores.



Business services, data services e services providers são Serviços Compartilhados Privados, por tanto, são registrados em um Private Service Registry. Estes serviços estarão disponíveis somente para outros serviços compartilhados

4.4 SOA Service Bus

- ❖ SOA requer um serviço para conectar todos os participantes
- ❖ As características principais do bus incluem:
 - Conectividade
 - Suporte às tecnologias heterogêneas
 - Suporte aos diferentes paradigmas de comunicação



Um elemento que não faz parte do conceito SOA mas contribui fortemente para a implementação é o *Enterprise Service Bus*, o qual conecta todos os participantes numa mesma plataforma.

Esta plataforma incorpora diferentes tecnologias como *message-oriented middleware* (MOM) como *MQSeries* e *WebServices*. Também permeia por padrões e provisão roteamento inteligente entre serviços e *endpoints* para estender as funcionalidades da companhia.

4.5 História do ESB

Nos primórdios da tecnologia, o ESB era concebido apenas para fazer os Web Services disponíveis para os *consumers*. Eles não abraçavam muitos padrões simplesmente porque muitos destes ainda não existiam.

Então de ESB eles tinham na verdade somente o nome.

A indústria amadureceu o entendimento sobre o papel que o mesmo poderia ocupar, ficou mais claro e hoje é muito mais que um simples “*service enabling*”.

Soluções para o mundo contemporâneo

A indústria de tecnologia está constantemente envolvida no entendimento dos problemas que as grandes empresas podem enfrentar no seu dia-a-dia. Os ESBs modernos são simplesmente uma das últimas ferramentas destinadas a auxiliar nessa empreitada.

Os benefícios reais da arquitetura SOA estão mudando a face das organizações e se os ESBs antigos não davam conta de todos os problemas, os atuais precisam resolver a maior parte deles.

4.5.1 Arquitetura Desacoplada

Muitos acreditam que os Web Services proporcionam desacoplamento entre os sistemas. Isso é parcialmente verdade. Web Services podem proporcionar algum tipo de desacoplamento, pois eles formalizam um contrato entre o *service consumer* e o *service provider*; modelo “**design by contract**” que também possui benefícios tangíveis. Entretanto você deve tomar cuidado na hora de criar um *Schema*, para que este seja neutro entre plataformas, proporcionando uma grande reutilização.

Contudo, se você der uma olhada mais a fundo no WSDL, verificará que os endpoints são retratados na descrição:

```
<service name="HelloWorldService">  
  <port binding = "s1>HelloWorldServiceSoapBinding  
    name = "HelloWorldPortSoapPort">  
    <s2:address  
      location=http://www.soaexpert.com.br:8001/esb>Hello World/>  
  </port>  
</service>
```

Por especificar a máquina e porta, ou conjunto de máquinas e portas, você está acoplando fortemente esse serviço a um computador

específico através de uma expressão. Você poderia até utilizar um servidor DNS para substituir partes da URL ao invés de apontar para máquinas diretamente, porém servidores DNS não são adequados para isso e não possuem habilidade para entender e administrar o status dos serviços que estão rodando nesses servidores.

Por tal motivo, arquitetura desacoplada não é totalmente alcançada pelo WSDL ou por Web Services sozinhos. Uma solução mais robusta é necessária para fazer a mediação entre *serviceclients* e *producers*, capaz de lidar com diferentes tipos de transportes e tecnologias de segurança.

Por exemplo: um serviço que pode ser invocado através do transporte HTTP, mas acionado por um serviço de baixo nível como JMS, ou e-mail , FTP, entre outros.

Esta abordagem consiste em traduzir – “wrap” ordens de serviço para os transportes apropriados dos clientes.

Oracle Servic Bus - Arquitetura Desacoplada

Em acordo com os benefícios da arquitetura desacoplada do WSDL e XSD, o OSB possui a habilidade para armazenar WSDL, XSD, XSLT e outros tipos que ele reconhece como “**resources**”- recursos.

Estes resources estão disponíveis através do cluster de OSB, permitindo a reutilização dos recursos sempre que necessário.

O benefício pode não estar claro ainda neste ponto do curso, mas se você considerar que muitas companhias definem seus modelos de domínio em XML – Canonical Data Model e o OSB pode guardar esses schemas XSD, você pode facilmente reutilizá-los numa gama enorme de WSDLs. Isso possibilitará a você criar um padrão de data types e message formats.

Transparência de localização

A transparência de localização é uma estratégia para ocultar a localização física dos *endpoints* dos *service clients*. Idealmente um *service client* deve saber apenas sobre uma máquina e *port name* para cada serviço. Esse cliente não precisa saber sobre os atuais *endpoints*.

Isto permite uma grande flexibilidade quando administrarmos nossos serviços. Você pode adicionar e remover service endpoints sem necessidade de recompilar seus service clients.

Mediação

Um Enterprise Service Bus é uma camada independente residindo entre o *service client* e os *service providers*. Essa camada adiciona um grande valor à arquitetura, já que possui a habilidade de realizar múltiplas operações: pode transformar um formato de dados para outro, de acordo com os schemas, enviar e receber inteligentemente de acordo com o conteúdo das mensagens, rotear para vários *service endpoints*, etc.

Schema Transformation

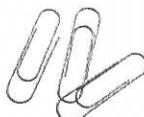
O serviço publicado no barramento pode utilizar um schema diferente do que o schema que o negócio representa. Essa é uma capacidade vital, especialmente quando fazemos agregação de serviços ou orquestração. Então o ESB permite que um service client possa receber dados usando um schema diferente do service provider.

A habilidade de transformar dados de um tipo para outro é crítica para o sucesso de qualquer ESB.

Service Aggregation

Service Aggregation atua como um *façade* e realiza uma série de chamadas aos Web Services, representando um único Web Service.

Ele atua como o pattern, fazendo múltiplas chamadas e retornando um único resultado.



A orquestração é bem parecida com a agregação, a diferença é que a orquestração inclui condições lógicas para saber qual serviço será invocado e sua ordem.

Load Balancing

Alguns produtos oferecem a capacidade de realizar o balanceamento de carga entre os múltiplos *service endpoints*. Quando se registra um serviço no ALSB, você pode informar uma lista de onde os serviços estão rodando e substituir a qualquer momento, sem necessidade de restart do servidor.

Reforço de Segurança

Você deve reforçar a segurança de uma maneira centralizada sempre que possível. O ESB lhe permite um grande nível de padronização e controle.

5.0 Como definir a granularidade do serviço

Existem várias técnicas de identificação de serviços e formação de processos de negócio no mercado. Uma das mais conhecidas e difundidas é o **SOAD** (Service Oriented Analysis and Design) criado pelo grupo de consultoria da IBM. Quando começamos o processo de identificação de serviços, seja enfatizando os processos de negócio e as suas atividades humanas ou sistêmicas que o compõem, ou enfatizando os bens e recursos que a TI possui que podem ser usados para a confecção dos serviços, enfrentamos vários desafios sobre como conceber serviços, do ponto de vista do SOA.

Serviços devem possuir algumas características mínimas, para que possam ser utilizados eficientemente dentro de soluções SOA. Um dos casos é a aplicação de reusabilidade nos serviços. Na prática, serviços que não oferecem um alto grau de reusabilidade, não são bons serviços, e fatalmente farão com que a solução como um todo tenda ao fracasso. Além do mais, serviços devem possuir interfaces bem definidas, baseadas em padrões abertos e largamente suportados.

Entretanto, nenhuma das características do SOA dão mais trabalho, do que o quanto completo e coeso o serviço é. A coesão e a completude dos serviços é um problema sério enfrentado por equipes de projeto de aplicações SOA, principalmente aquelas equipes menos experimentadas nas técnicas de análise e desenho orientado a objetos.

A granularidade pode ser dividida em duas partes, sendo:

Granularidade fina (***fine-grained***) e granularidade grossa (***coarse-grained***), onde granularidade fina determina que precisamos de muitos "grãos", enquanto na granularidade grossa, teremos poucos "grãos", e bem maiores.

Com a granularidade fina, teremos serviços com poucas operações, mas dividiremos essas operações por vários serviços. Já com a granularidade grossa, isso se inverte, ou seja, teremos poucos serviços, mas cada um deles conterá uma porção bem maior de operações.

Cada uma das técnicas tem suas vantagens e desvantagens, quando definimos uma granularidade muito fina, temos pequenos "blocos" de funcionalidades bem específicas e muitas vezes independentes, e que ficam bem mais fáceis de serem atualizadas, distribuídas e gerenciadas, mas isso pode se tornar complexo demais para aqueles que consomem, já que terão que compor e sincronizar suas operações, para que atinja um determinado objetivo. Por outro lado, a granularidade grossa pode tornar os serviços mais auto-suficientes, e o problema disso é que cada serviço poderá, acidentalmente fazer muito mais trabalho do que ele realmente deveria.

A primeira coisa a ser feita, na hora da definição de serviços, é decidir corretamente sobre qual o grau correto de granularidade. Isso se dá de duas formas: Sobre quantos argumentos um método deve possuir de forma que ele seja mais reutilizável e sobre quantos métodos de negócio, determinado serviço deva possuir, sem afetar a reusabilidade e coesão do mesmo.

Bom, para lidar com este dilema, é importante que seja aplicado no mínimo, algumas técnicas de OOAD padrões. Em projetos OO, normalmente os objetos são concebidos através de análises intensas sobre substantivos. Uma técnica interessante sobre como descobrir classes e objetos, é o uso dos cartões CRC. Nesta técnica, classes são encontradas fazendo análises de substantivos. A partir do momento que os substantivos (ou classes) são encontrados, inicia-se a definição dos principais verbos desta classe.

Da mesma forma que investigamos classes em sistemas OO, devemos pensar em serviços. Em regra, os serviços devem ser identificados da mesma forma. Os métodos destes serviços também devem ser identificados da mesma forma. A grande questão, é que quando falamos de serviços, devemos pensar em outros fatores oriundos do mundo SOA.

Quando criamos sistemas OO, normalmente tendemos a criar um amontoado de classes e coesas e como suas responsabilidades. Isso leva a criação de muitas classes no sistema (sistemas grandes têm em média de 500 a 1000 classes). No mundo SOA, isso não deve ser seguido a risca, pois devemos ter poucos serviços, porém coesos e completos. O motivo por trás de ter poucos serviços, é que cada serviço representa um recurso de rede que vai ser localizado. Os serviços tendem a ser em menor número, para diminuir as chamadas remotas na rede, prevendo a redução do uso da largura de banda.

Neste caso, entramos no dilema: Como criar serviços como poucas responsabilidades (prevendo o problema de latência de rede) mas ainda fazendo com que o mesmo seja reutilizável e coeso, de forma a integrar-se mais facilmente nos processos de negócio das corporações? A resposta pra isso é composição!

5.1 Exercício de Modelagem

Imagine um serviço de Processamento de Pedidos. Defina suas operações baseadas em verbos. Por exemplo, imagine uma classe concebida chamada 'ProcessadorCartão'.

A partir desta classe, devemos pensar sobre quais operações ou responsabilidades, ela terá. Podemos pensar em verbos como: 'verificarCartão', 'verificarValidade', 'efetuarCompra', 'realizarSaque' etc. Neste caso, observamos quais são as responsabilidades relativas a classe, que de alguma forma fazem sentido pra ela, e estão dentro do contexto da mesma.

OBS: Qualquer verbo que estiver fora do seu contexto (como 'consultarPreço' por exemplo) estaria violando as regras de coesão desta classe.

5.2 DataCentric

Outro grande ponto a ser considerado na construção de serviços, é a criação com interfaces CRUD (Create, Read, Update e Delete). A proposta destes tipos de serviços é permitir, na maioria das vezes, a manipulação de registros dentro de uma determinada base de dados.

Nestes casos, o serviço será apenas uma espécie de wrapper para os dados, não fazendo nada além do que as operações básicas que todo banco de dados possui (INSERT, SELECT, UPDATE e DELETE).

Quando você trabalha com uma aplicação data-centric, onde a toda a regra se concentra em manipular a base de dados, talvez esses tipos de serviços sejam úteis.

Imagine um novo cliente que deseja abrir uma conta bancária em um determinado banco, com um cartão de crédito vinculado. O processo de cadastro do cliente consistirá em:

1. Validar os dados, como idade, renda, endereço, etc.;
2. Consultar outras instituições financeiras para se certificar de que ele é um bom pagador
3. Definir o limite que ele terá no cheque especial
4. Inserir o cliente na base de dados
5. Criar a conta corrente para este cliente
6. Efetuar o lançamento da taxa de cadastro/abertura na conta corrente recém criada
7. Comunicar com o serviço de cartões de crédito, para que ele gere um novo cartão para este cliente
8. Notificar outros departamentos do banco de que uma nova conta foi aberta, para oferecimento de novos produtos.

O cadastro de um novo cliente não consiste apenas em adicioná-lo na base de dados. Há muito mais do que isso. Se modelarmos nossos serviços orientado à dados, eu teria um serviço que manipula os clientes, outro serviço que manipula a conta corrente, outro de notificação e por aí vai. Quanto mais entidades/tabelas você tiver envolvidos em uma mesma tarefa, mais complicado ficará para gerenciar tudo isso, principalmente do ponto de vista daquele que consumirá esses serviços.

O consumo de serviço é um processo caro para se fazer à todo momento, e neste cenário, para efetuar o cadastro de um cliente, eu precisarei chamar, no mínimo, quatro serviços e tudo o que eu precisarei passar para eles, é exatamente os dados do cliente que eu desejo avaliar/cadastrar. Outro ponto importante é com relação ao fluxo de informações. Neste caso, fica sempre sob responsabilidade do cliente que consome esses serviços, configurar a ordem de chamadas, e em um ambiente onde múltiplas aplicações podem incluir clientes, eventualmente uma delas poderá alterar essa ordem, fazendo com que o processo fique em um estado inválido, comprometendo assim a veracidade e consistência das informações. Nada impedirá que uma pessoa maliciosa invoque apenas o serviço de cadastro de cliente diretamente, sem passar pelas políticas de validação necessárias.

Quando temos várias "sub-tarefas" que se juntam para algo maior, em muitos casos queremos garantir a atomicidade, que garantirá que todos os passos sejam efetuados com sucesso, ou tudo falhará. O que garante a atomicidade são as transações, e transações distribuídas são caras, e todas as chamadas para esses serviços devem estar envolvidas dentro dessa transação, que será, também, coordenada pelo cliente, que será o responsável por avaliar se tudo deu certo. Se sim, ele efetivará (**Commit**), do contrário, irá desfazer (**Rollback**).

Um segundo cenário que também ilustra isso: você possui clientes e cada um deles possui um *flag* que determina a situação dele dentro da sua empresa: *Ativo*, *Bloqueado*, *EmProcessoJuridico*, etc. Da mesma forma que vimos antes, alterar situação dele vai muito além de um simples comando de UPDATE na base de dados. Quando eu mover um determinado cliente para a situação de *EmProcessoJuridico*, eu terei que inserir um item no histórico deste cliente, desativar algumas opções que o mesmo tem, e alterar a sua situação na tabela do banco de dados. Se movê-lo para *Bloqueado*, terei que inserir um item no seu histórico, efetuar um *lockdown* em todas as contas de acesso desse cliente no site, notificar o gerente responsável e, finalmente, efetuar o UPDATE da coluna onde armazeno a situação atual na tabela de clientes.

Como podemos perceber, interfaces CRUD definem os serviços como sendo ***data-centric***, que modelando dessa forma, nós perderemos o contexto de negócio que será executado pelo cliente, tornando bem mais complicado de se entender o processo como um todo. Ao modelar os serviços, o ideal seria pensar em ***task-centric***, ou seja, o serviço fornecerá operações que englobam grande parte do processo, não sendo o consumidor o responsável por isso. Nos dois cenários que vimos acima, teríamos um serviço de cliente, e que me forneceria uma operação para criar um novo cliente dentro do banco (*IncluirNovoCliente*), outra operação para bloquear o cliente (*Bloquear*), outro para criar um processo contra este cliente (*AbrirProcessoJuridico*), que o moverá para a situação *EmProcessoJuridico*, e assim por diante.

Os **Entity Services** levam o nome de uma entidade, como por exemplo: Cliente, ContaCorrente, PoliticasDeValidacao, etc., não fazendo nada além do que o nome diz, ou seja, disponibilizará apenas as informações referentes à respectiva entidade, não conhecendo nada sobre negócios. Já os nomes dos Task Services são voltados para o negócio em si: AdministracaoDeClientes, GestorDeCredito, CobrancaDeTitulos, etc. E ainda, os Task Services poderão utilizar um ou vários Entity Services para executar uma determinada tarefa, atentando-se sempre aos conceitos que vimos acima, como é o caso do baixo/alto acoplamento e a granularidade.

5.3 Modelagem de Serviços e Contratos WSDL

Nesse momento vamos aprender como criar um contrato WSDL na mão. Esta é uma abordagem bastante diferente para quem está acostumado a escrever Web Services em Java, que gera automaticamente.

Devemos lembrar que a geração automática não é recomendada, podendo prendê-lo a uma única plataforma.

De um modo geral, o uso de conversores traz alguns problemas de compatibilidade, como Basic-to-C converter, o qual não funcionava como esperado mas trazia algum auxílio para que os programadores Basic entendessem um pouco mais sobre a sintaxe do C.

Construindo o WSDL

Basicamente um WSDL é constituído de seis partes, contidas no elemento root `<definitions>`.

1. **types**: define a estrutura de dados utilizada em seu WSDL. A estrutura de dados é expressa como um *schema XML – XSD*.
2. **portType**: essa é a interface, que, como no java, define a implementação do seu serviço.
3. **message**: aqui é definido o formato da mensagem (pense em documentos) que são utilizados pelos web services.
4. **binding**: descreve como a portType é mapeada numa expressão concreta de estrutura de dados e protocolos.

5. **port**: descreve o endereço do endpoint do serviço, isto é, onde o serviço pode ser encontrado.

6. **service**: uma coleção de elementos port, assim você pode especificar que um serviço pode ser encontrado em diversos lugares.

Exemplificando o WSDL

O código abaixo mostra um exemplo de um simples WSDL que define um serviço de consulta cliente, e retorna o mesmo através do seu ID.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
    xmlns:cts="http://www.soaexpert.com.br/CustomerService/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="CustomerService"
    targetNamespace="http://www.soaexpert.com.br/CustomerService/">
    <wsdl:types>
        <xsd:schema
            targetNamespace="http://www.soaexpert.com.br/CustomerService/">
            <xsd:element name="findCustomer">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="id" type="xsd:int" />
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="findCustomerResponse">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="name" type="xsd:string" />
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:schema>
    </wsdl:types>
    <wsdl:message name="findCustomerRequest">
        <wsdl:part element="cts:findCustomer" name="parameters" />
    </wsdl:message>
    <wsdl:message name="findCustomerResponse">
        <wsdl:part element="cts:findCustomerResponse" name="parameters" />
    </wsdl:message>
    <wsdl:portType name="CustomerService">
        <wsdl:operation name="findCustomer">
            <wsdl:input message="cts:findCustomerRequest" />
            <wsdl:output message="cts:findCustomerResponse" />
        </wsdl:operation>
    </wsdl:portType>
```

6. Namespaces

Antes de começarmos a discutir sobre WSDL mais a fundo, introduziremos alguns conceitos sobre XML namespaces.

O namespace é um conceito usado intensivamente tanto pelo WSDL quanto pelas estruturas de dados XML – XSD Schemas.

Namespaces podem ser difíceis para ler e para entender, ao menos que você entenda perfeitamente seu papel e como ele afeta o documento.

Um namespace é uma forma de categorizar ou agrupar elementos, estrutura de dados e atributos em um XML.

Para fazer um efeito comparativo, é análogo ao *package* do Java ou a *keyword namespace* do C#.

No WSDL acima, o *namespace* instancia uma variável, **CTS**, a qual atua como ponteiro para referência do mesmo. Você poderia, por exemplo, ter duas ou mais referências para o mesmo namespace, utilizando prefixos diferentes.

```
xmlns:cts="http://www.soaexpert.com.br/CustomerService/"
```

Default Namespace

Todo elemento num documento XML ou XSD pertence a um namespace.

O *namespace* padrão é aplicado a todos os nós num documento que não possuem um *namespace* explicitado.

Definir um *namespace* padrão é similar à definir um *namespace* com um prefixo, você apenas **não** define o prefixo. Entretanto poderá existir somente um *namespace* padrão para cada elemento.

6.1 Types

WSDL e XML Schemas definem estrutura de dados.

Native Data Types

XML Schema provisiona um grande número de tipos de dados nativo, também conhecidos como primitivos, que você pode fazer uso facilmente nas suas estruturas – strings, integers, dates, times, e muitos outros.

Exemplo de declaração de um objeto do tipo string:

```
<element name="MyString" type="string"/>
```

Custom Data Types

Elevando um pouco o nível, você pode utilizar as estruturas primitivas para definir um objeto mais complexo, como cliente:

```
<complexType name="Cliente">
    <sequence>
        <element name="idCliente" type = "int" minOccurs=1/>
        <element name = "primeiroNome" type="String minOccurs"1"/>
    </sequence>
</complexType>
```

Importando XML Schemas

Uma das melhores maneiras de promover a reusabilidade e centralizar as definições do modelo de domínios é importar uma estrutura (schema) para seu WSDL, que é feito dentro do elemento **<types>**

```
<import namespace=" [a uri do namespace]" schemaLocation= "[caminho do seu arquivo .xsd]"/>
```

Message

Uma mensagem descreve de forma abstrata **input**, **output** ou **fault** messages (falhas).

Mensagens são compostas de um ou mais **<part>** element e estes descrevem a composição da **<message>**.

Utilizando a abordagem document-centric num WSDL, os elementos `part` de uma message se referem à estrutura de dados que o atributo `element` está utilizando.

portType

A sessão `portType` de um WSDL descreve a interface do web service. Essa é análoga à interface Java, pois define num alto-nível como a operação do serviço funciona, ou seja, quais argumentos são esperados e quais resultados são retornados.

Binding

É utilizado para definir como um `portType` é atrelado a um protocolo de transporte e o tipo de encoding.

O mais comumente utilizado é a combinação de SOAP sob HTTP, mas outras combinações são possíveis, como HTTP/POST e HTTP/GET especialmente quando você está interagindo com Web services criados antes que o padrão HTTP/SOAP fosse tão popular.

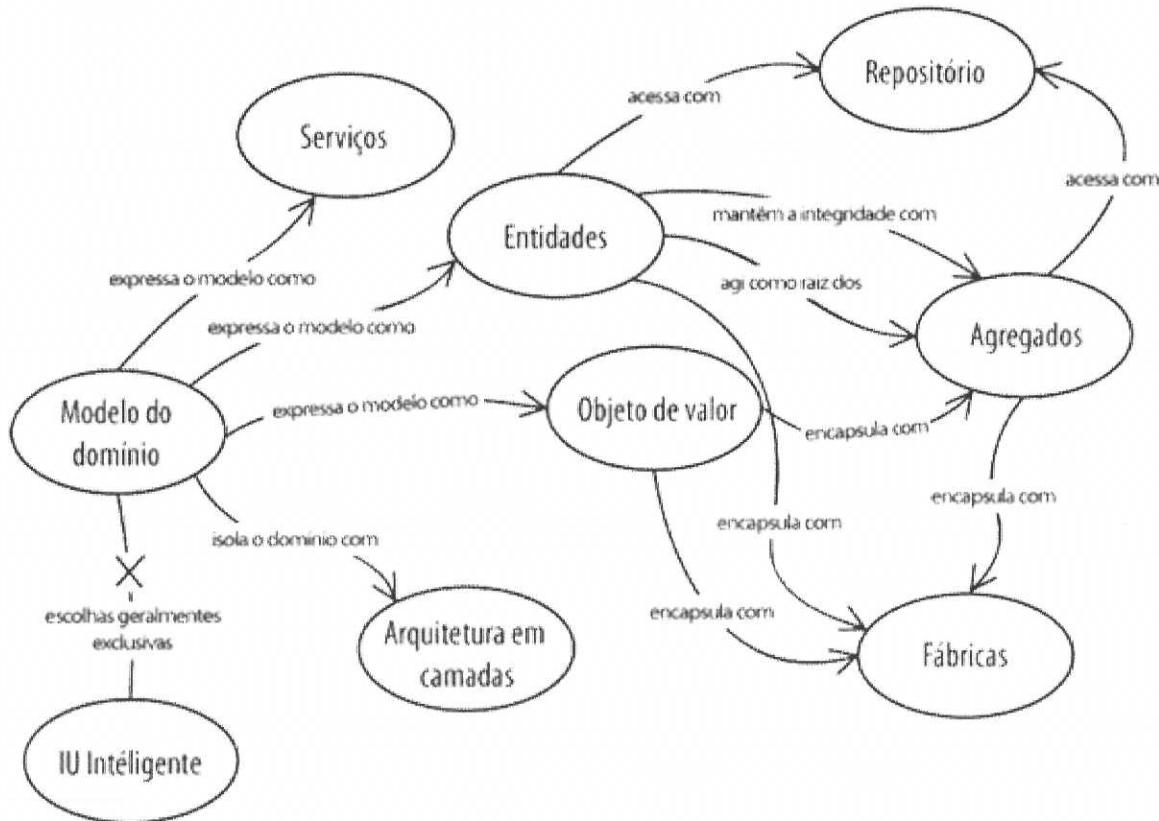
Boas práticas :

Procure definir separadamente seus tipos em arquivos XML distintos e os refencie no seu WSDL.

Conforme já mencionado, um XML namespace é análogo a um Java package. Entretanto o package está no escopo da aplicação e o namespace está no escopo da corporação.

7.0 Domain-Driven Design

“Software é uma arte de design, que como qualquer outra arte, não pode ser estudada e aprendida como uma ciência exata, por teoremas e fórmulas.” - **Floyd Marinescu**



O que é DDD ?

Domain-Driven Design não é uma tecnologia ou metodologia, mas sim uma abordagem de design de software disciplinada que reúne um conjunto de conceitos, técnicas e “princípios” com o foco no **domínio** e na lógica do domínio para criar um **Domain Model**.

- É um conjunto de práticas, com técnicas de modelagem comprovadas
- Ideal para domínios complexos
- Um conjunto de padrões (patterns) que suporta uma clara e coerente visualização do modelo de domínio
- Estratégias “pragmáticas” que permitem a aplicação escalar em tamanho e complexidade, mantendo a integridade
- Linguagem onipresente

Não investir em um modelo de domínio pode levar à uma arquitetura de aplicativo com uma "**Fat Service Layer**" e "**Anemic Domain Model**", onde as classes de fachada (geralmente Stateless Session Beans) começar a acumular cada vez mais lógica de negócios e tornam os domínio (Pojos) meros objetos de dados com *getters* e *setters*.

Esta abordagem também leva a lógica de negócios e regras específicas, a serem distribuída em várias classes de fachada (Facade) diferentes.

7.1 Model

Um modelo é uma representação de um domínio servindo a um propósito. Não existe um modelo “perfeito” para um domínio e sim uma adaptação para o seu propósito.

Expressando o Model

O modelo e o entendimento de design devem estar sincronizados e a codificação é a última maneira de se expressar um modelo. Artefatos intermediários, como diagramas e documentações, servem para apoiar temporariamente a abstração.

Melhorando o design:

Uma forma de melhorar o design é trabalhar sempre próximo aos convededores do domínio – “Domain Experts”.

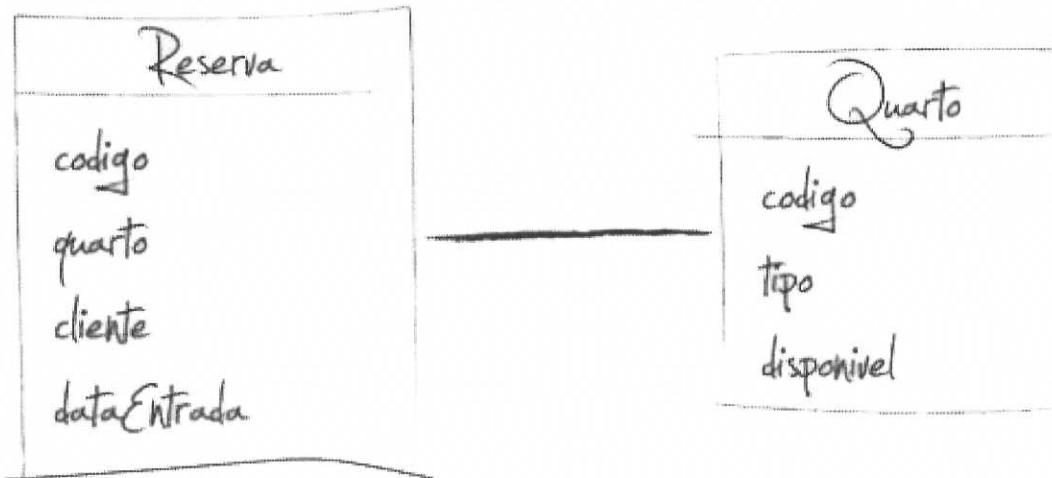
A equipe de desenvolvimento e os *Domain Experts* devem continuamente refinar o modelo e melhorar o entendimento sobre o domínio, e para que isso seja de fato executado canais de comunicação dentro da companhia devem ser criados.

Na essência, o trabalho de design de software que irá completar um processo, consiste em traduzir o cenário de negocio (domínio) e traduzir em algo de utilidade, que realmente seja usado pela empresa.

7.2 Ubiquitous Language

O conhecimento sobre o domínio é expressado numa “**linguagem ubiquoa**”, permitindo aos *domain experts*, designers, programadores, compartilharem a mesma visão.

A ideia da linguagem é que não haja “ambiguidade” entre todas as partes envolvidas no processo.



Resumindo:

É a linguagem baseada em modelos utilizada pelos *experts* em **domínio**.

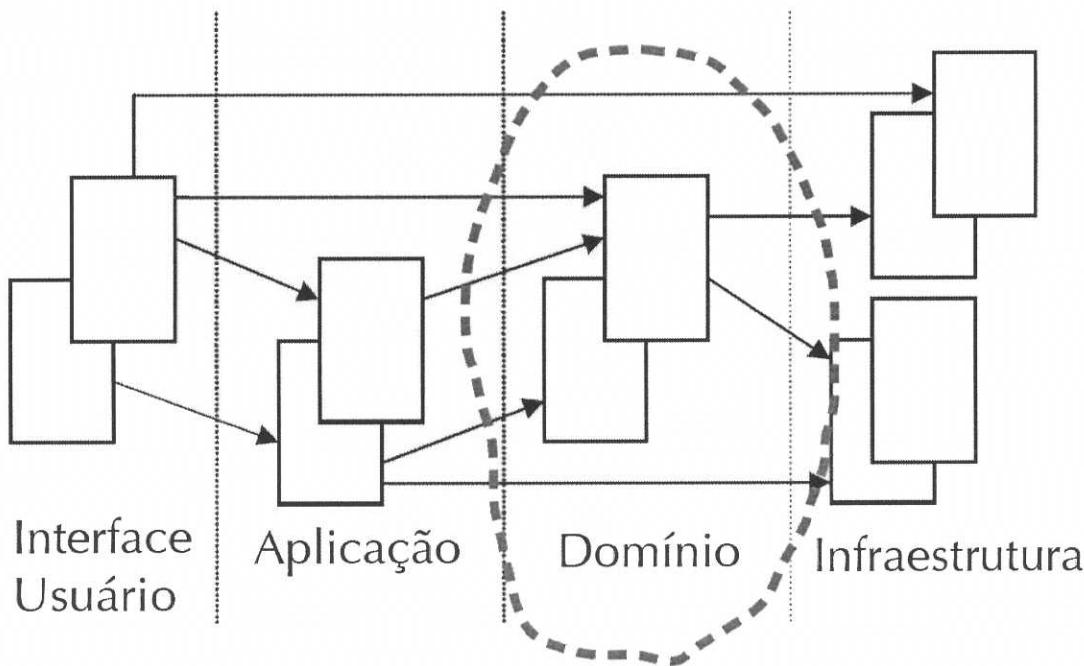
- Substantivos == Classes
- Verbos == Métodos, serviços e etc.

Exemplo: Um especialista em RH pode postar Empregos, para o Classificados.

Classes - Emprego, Classificados

Ações: Classificados.postarEmprego(Emprego)

7.3 Arquitetura em Camadas



É uma abstração do problema real, desenvolvida em conjunto pelos especialistas do domínio e desenvolvedores. No DDD, é chamado de **Domain Model**.

É esse modelo que os desenvolvedores vão implementar em código, literalmente (Item por item, como foi acordado por todos). Será desenvolvido um código limpo, com palavras do domínio que representa, na programação, o domínio em discussão.

Usando DDD, seu programa orientado a objetos deve expressar a riqueza do *domain model*. Qualquer mudança no modelo, e acredite isso é muito comum, deve ser refletida imediatamente no código. Se algo do modelo torna-se inviável de se implementar tecnicamente, não se faz um “ajuste” no código; o modelo deve ser mudado para ser mais fácil de se implementar.

Resumo: Seu código será expressão do **modelo**, que por sua vez é baseado totalmente no **domínio**.

7.4 Patterns

O DDD define uma série de *patterns* (padrões) para facilitar a implementação do modelo em código.

Entities

Possui **identificação** única e um **estado**, entretanto elas não lidam diretamente com questões de persistência de informações.

Suas responsabilidades e associações são baseadas na sua identificação e não em seus atributos.

Value Objects

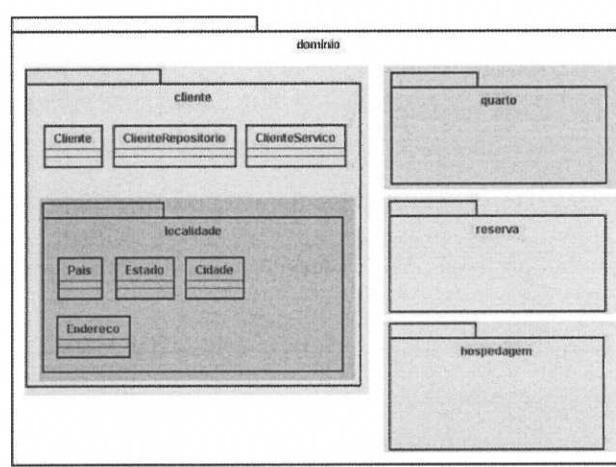
É um objeto simples, **sem identificação** (id), que serve para transporte de informações, por tanto, imutável e pode ser compartilhado por diferentes Entities.

Exemplos: Money, Endereço, Códigos etc.

Services

Um serviço representa uma operação de um domain model. Esta é sem estado (stateless) e pertence à camada **Domain Layer**.

Modules



Módulos são um meio de partitionar o sistema para torná-lo mais gerenciável. Muitas linguagens implementam o conceito através de **packages**.

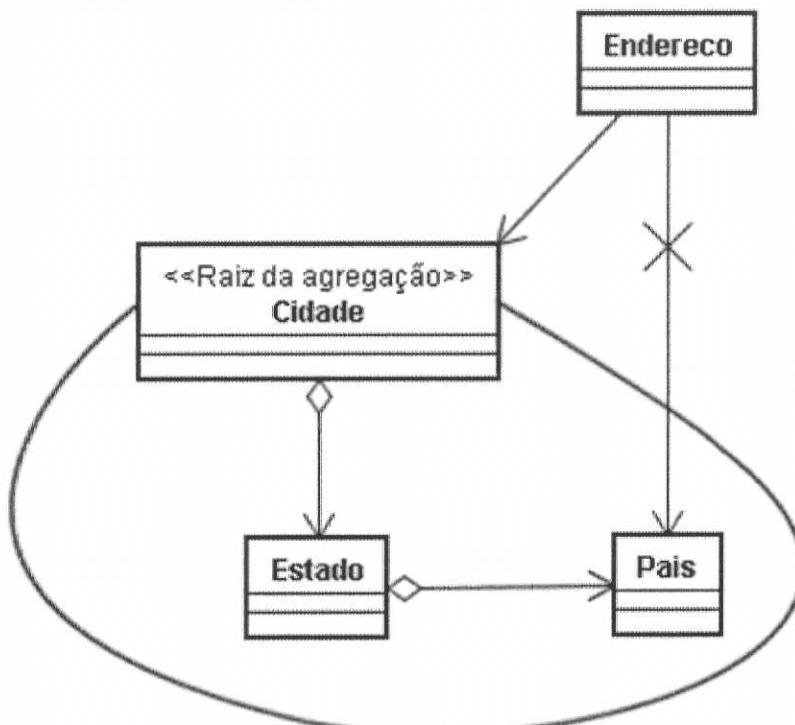
No Domain-Driven Design, packages devem ser divididos de acordo com o propósito intrínseco a cada tipo de objeto.

Os tipos de objeto devem resultar numa convenção clara e evidente
– **Naming Convention**.

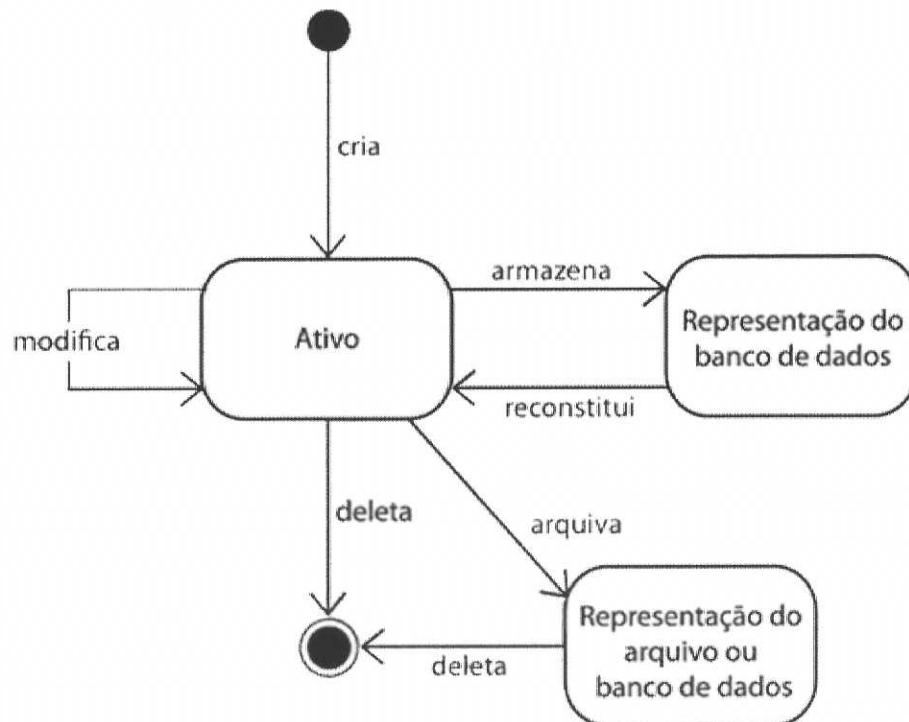
Aggregates

É um agrupamento de objetos associados tratados como uma única peça para o propósito de mudança de dados.

Uma Entity específica é selecionada como Raiz (root) do agrupamento.



Patterns de criação de objetos



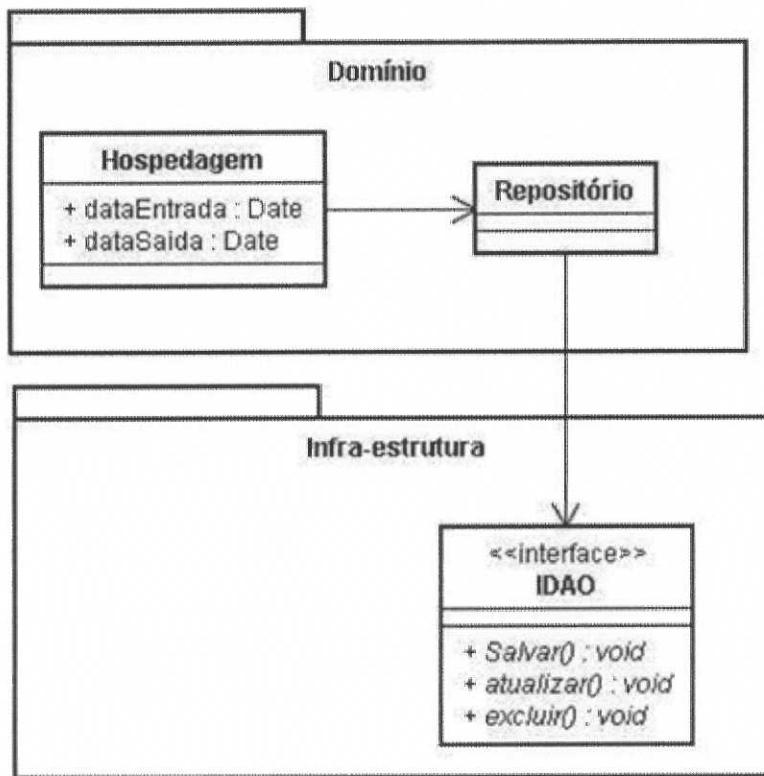
Factory

Os patterns de criação de objetos ajudam a criar as Entities e Value Objects de modo que eles sejam sempre válidos.

O seu papel é gerenciar o ciclo de vida, encapsulamento da complexidade, por exemplo, de objetos complexos e agregações.

É importante lembrar que em DDD, a responsabilidade de criação de uma instância “complexa”, não pertence ao objeto em si.

Repositories



Os Repositories provisionam a capacidade de acessar os dados de forma transparente, como se fosse uma coleção em memória.

Eles oferecem uma camada de persistência subjacente, oferecida como serviços nos termos dos elementos do domain model.

8.0 XML e DTD Fundamentos

O que é XML ?

Extensible Markup Language (XML) é linguagem de marcação de dados (meta-markup language) que provê um formato para descrever dados estruturados.

Um elemento XML pode ter dados declarados como sendo preços de venda, taxas de preço, um título de livro, a quantidade de chuva, ou qualquer outro tipo de elemento de dado.

8.1 XML Origem

HTML e XML são primos. Eles derivam da mesma inspiração, o SGML. Ambos identificam elementos em uma página e ambos utilizam sintaxes similares. A grande diferença entre HTML e XML é que o HTML descreve a aparência e a ações em uma página na rede, enquanto o XML o que cada trecho de dados é ou representa.

Resumindo: O XML descreve o conteúdo do documento !

Como o **HTML**, o XML também faz uso de **tags** (palavras encapsuladas por sinais '<' e '>') e atributos (definidos com `name="value"`), mas enquanto o **HTML** especifica cada sentido para as tags e atributos, o XML usa as **tags** somente para delimitar trechos de dados e deixa a interpretação do dado a ser realizada completamente para a aplicação que o está lendo.

Resumindo, enquanto em um documento HTML uma tag `<p>` indica um parágrafo, no XML essa tag pode indicar um preço, um parâmetro, uma pessoa, ou qualquer outra coisa que se possa imaginar (inclusive algo que não tenha nada a ver com um p como por exemplo autores de livros).

Os arquivos XML são arquivos texto, mas não são tão destinados à leitura por um ser humano como o HTML é. Os documentos XML são arquivos texto porque facilitam que os programadores ou desenvolvedores "debuguem" mais facilmente as aplicações, de forma que um simples editor de textos pode ser usado para corrigir um erro em um arquivo XML. Mas as regras de formatação para documentos XML são muito mais rígidas do que para documentos HTML.

Uma tag esquecida ou um atributo sem aspas torna o documento inutilizável, enquanto que no HTML isso é tolerado. As especificações oficiais do XML determinam que as aplicações não podem tentar adivinhar o que está errado em um arquivo (no HTML isso acontece), mas sim devem parar de interpretá-lo e reportar o erro.

8.2 Características do XML

O XML provê um padrão que pode codificar o conteúdo, as semânticas e as esquematizações para uma grande variedade de aplicações desde simples até as mais complexas, dentre elas:

- Um simples documento
- Um registro estruturado tal como uma ordem de compra de produtos
- Um objeto com métodos e dados como objetos Java
- Um registro de dados. Um exemplo seria o resultado de uma consulta a bancos de dados
- Apresentação gráfica, como interface de aplicações de usuário
- Entidades e tipos de esquema padrões

Uma característica importante é a manipulação de dados sem a necessidade de relacionar com o servidor. Dessa forma, os servidores tem menor sobrecarga, reduzindo a necessidade de computação e reduzindo também a requisição de banda pna comunicação cliente e servidor.

O XML é considerado de grande importância numa arquitetura heterogênea, pois provê a capacidade de interoperabilidade das diferentes plataformas e tem um padrão flexível, aberto e independente.

8.3 Estrutura do XML

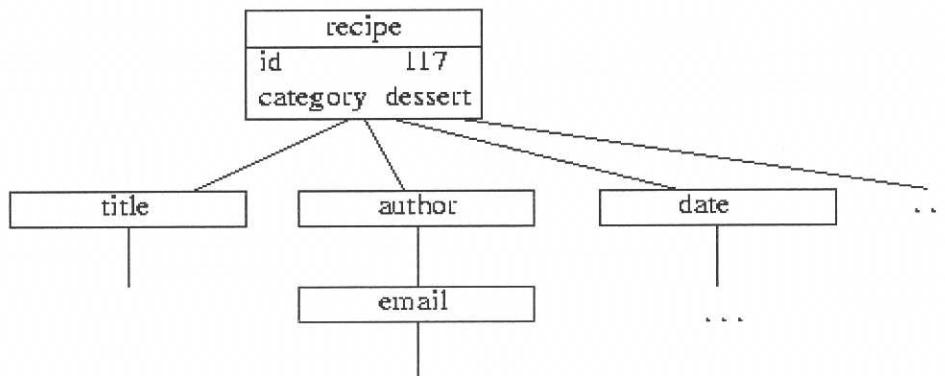
Um documento XML é uma árvore rotulada onde um nó externo consiste de:

- **dados de caracteres** (uma sequência de texto).
- **instruções de processamento** (anotações para os processadores), tipicamente no cabeçalho do documento.
- **um comentário** (nunca com semântica acompanhando).
- **uma declaração de entidade** (simples macros) nós DTD (Document Type Declaration).

Um nó interno é um elemento, o qual é rotulado com:

- Um nome ou um conjunto de atributos, cada qual consistindo de um nome e um valor.

Normalmente, comentários, declarações de entidades e informações DTD não são explicitamente representadas na árvore.



8.4 Estrutura de uma árvore XML.

Geralmente, a árvore tem um nó raiz especial acima do elemento raiz.

Tags

Um documento XML é um texto (em formato Unicode) com tags de marcação (markup tags) e outras informações.

As markup tags possuem a seguinte estrutura:

...<bla attr="val" ...>...</bla>...

| | ||

| | | uma tag finalizadora de elemento

| | | o contexto do elemento

| | um atributo com nome attr e valor val, com valores delimitados por ' ' ou "

uma tag inicializadora de elemento com nome bla

Notação para elementos vazios: ...<bla attr="val" .../>...

Os documentos XML são sensíveis à letras maiúsculas e minúsculas. Um documento XML é bem formatado quando segue algumas regras básicas. Tais regras são mais simples do que para documentos HTML e permitem que os dados sejam lidos e expostos sem nenhuma descrição externa ou conhecimento do sentido dos dados XML.

Documentos bem estruturados:

- têm casamentos das tags de **início** e **fim**;
- as tags de elemento têm que ser apropriadamente posicionadas ; e
- os elementos não podem se sobrepor.

Um exemplo de sobreposição é o seguinte :

```
<title>Descrição dos diversos modelos de carros<sub> da marca Ford</title> Fabio Poletto</sub>
```

E, corrigindo o erro:

```
<title>Descrição dos diversos modelos de carros <sub> da marca Ford</sub><br/><author> Fabio Poletto</author> </title>
```

8.5 Declarações de Tipos de Elementos DTD

Declarações de tipos de elementos identificam os nomes dos elementos e a natureza do seu conteúdo. Uma declaração de tipo de elemento típica se parece com isto:

```
<!ELEMENT piada (João+, José, aplausos?)>
```

Esta declaração identifica o elemento nomeado como **piada**. Seu *modelo de conteúdo* segue o nome do elemento. O modelo de conteúdo define o que um elemento pode conter. Neste caso, uma **piada** deve conter **João** e **José** e pode conter **aplausos**. As vírgulas entre os nomes dos elementos indicam que eles devem ocorrer em sucessão. O sinal de adição após **João** indica que ele pode ser repetido mais de uma vez, mas deve ocorrer pelo menos uma vez. O ponto de interrogação após **aplausos** indica que ele é opcional (pode estar ausente ou ocorrer somente uma vez).

Um nome sem pontuação, como **José**, deve ocorrer somente uma vez. As declarações para todos os elementos usados em qualquer modelo de conteúdo devem estar presentes para que um processador XML verifique a validade do documento.

Além dos nomes de elementos, o símbolo especial **#PCDATA** é reservado para indicar dados de caracter. A cláusula **PCDATA** significa dado de caracter analisável.

Os elementos que contêm somente outros elementos são ditos que têm *conteúdo de elementos*. Os elementos que contêm outros elementos e **#PCDATA** são ditos que têm *conteúdo misturado*.

Por exemplo, a definição para **José** pode ser
`<!ELEMENT José (#PCDATA | citação)*>`

A barra vertical indica um relacionamento "ou" e o asterisco indica que o conteúdo é opcional (pode ocorrer zero ou mais vezes); por esta definição, portanto, **José** pode conter zero ou mais caracteres e marcas **citação**, misturadas em qualquer ordem. Todos os modelos de conteúdo misturado devem ter esta forma: **#PCDATA** deve vir primeiro, todos os elementos devem ser separados por barras verticais e o grupo inteiro deve ser opcional.

Outros dois modelos de conteúdo são possíveis: **EMPTY** indica que o elemento não possui conteúdo (e, consequentemente, não tem marca de término) e **ANY** indica que *qualquer* conteúdo é permitido. O modelo de conteúdo **ANY** é algumas vezes útil durante a conversão de documentos, mas deveria ser evitado ao máximo em um ambiente de produção, pois desabilita toda a verificação do conteúdo deste elemento. Aqui está um conjunto completo das declarações de elementos para o Exemplo 1.

Exemplo 2: declarações de elementos para Exemplo 1

```
<!ELEMENT piada (João+, José, aplausos?)>
<!ELEMENT João (#PCDATA | citação)*>
<!ELEMENT José (#PCDATA | citação)*>
<!ELEMENT citação (#PCDATA)*>
<!ELEMENT aplausos EMPTY>
```

8.6 Declarações de Listas de Atributos DTD

Declarações de listas de atributos identificam que elementos podem ter atributos, que atributos eles podem ter, que valores os atributos podem suportar e qual valor é o padrão. Uma declaração de lista de atributos típica se parece com isto:

```
<!ATTLIST piada
  nome
  ID
  #REQUIRED
  rótulo
  CDATA
  #IMPLIED
  estado ( engraçada | nãoengraçada ) 'engraçada'>
```

Neste exemplo, o elemento **piada** possui três atributos: **nome**, que é um ID e é obrigatório; **rótulo**, que é uma cadeia de caracteres (dados de caracter) e não é obrigatório; e **estado**, que deve ser ou **engraçada** ou **nãoengraçada** e por padrão é **engraçada**, se nenhum valor é especificado.

Cada atributo em uma declaração tem três partes: um nome, um tipo e um valor padrão. Você tem liberdade para selecionar qualquer nome desejado, sujeito a algumas pequenas restrições, mas os nomes não podem ser repetidos no mesmo elemento.

Existem 5 tipos de atributos possíveis:

1. CDATA

Atributos CDATA são cadeias de caracteres; qualquer texto é permitido. Não confunda atributos CDATA com seções CDATA; eles não têm relação.

2. ID

O valor de um atributo ID deve ser um nome. Todos os valores usados para IDs em um documento devem ser diferentes. Os IDs identificam unicamente elementos individuais em um documento.

Os elementos podem ter um único atributo ID.

3. IDREF ou IDREFS

O valor de um atributo IDREF deve ser o valor de um único atributo ID em algum elemento no documento. O valor de um atributo IDREFS pode conter valores IDREF múltiplos separados por espaços em branco.

4. ENTITY ou ENTITIES

O valor de um atributo ENTITY deve ser o nome de uma única entidade (veja sobre declarações de entidades abaixo). O valor de um atributo ENTITIES pode conter valores de entidades múltiplos separados por espaços em branco.

5. NMTOKEN ou NMTOKENS

Atributos de símbolos de nome são uma forma restrita do atributo de cadeia de caracteres. Em geral, um atributo NMTOKEN deve consistir de uma única palavra, mas não há restrições adicionais para a palavra; não tem que estar associado com outro atributo ou declaração. O valor de um atributo NMTOKENS pode conter valores NMTOKEN múltiplos separados por espaços em branco

Uma lista de nomes

Você pode especificar que o valor de um atributo deve ser pego de uma lista específica de nomes. Isto é freqüentemente chamado de tipo enumerado, porque cada um dos valores possíveis está explicitamente enumerado na declaração. Alternativamente, você pode especificar que os nomes devem atender a um nome de notação (veja sobre declarações de notação abaixo).

Há quatro valores padrão possíveis:

#REQUIRED

O atributo deve ter um valor explicitamente especificado em cada ocorrência do elemento no documento.

#IMPLIED

O valor do atributo não é requerido, e nenhum valor padrão é fornecido. Se um valor não é especificado, o processador XML deve proceder sem um.

"valor"

Qualquer valor válido pode ser dado a um atributo como padrão. O valor do atributo não é requerido em cada elemento no documento, e se ele estiver presente, será dado a ele o valor padrão.

#FIXED**"value"**

Uma declaração de atributo pode especificar que um atributo tem um valor fixo. Neste caso, o atributo não é requerido, mas se ele ocorrer deve ter o valor especificado. Se não estiver presente, será dado a ele o valor padrão.

8.7 Declarações de Entidades DTD

Declarações de entidades lhe permitem associar um nome com algum outro fragmento de conteúdo. Essa construção pode ser um pedaço de texto normal, um pedaço de uma declaração de tipo de documento ou uma referência a um arquivo externo que contém texto ou dados binários.

Declarações de entidades típicas são mostradas no Exemplo 3.

Exemplo3 : Declaração de entidades típica

```
<!ENTITY
ATI
"SOAEXPERT, Inc.">

<!ENTITY boilerplate      SYSTEM
"/standard/legalnotice.xml">

<!ENTITY ATIlogo
SYSTEM "/standard/logo.gif" NDATA GIF87A>
```

Existem três tipos de entidades:

Entidades Internas

Entidades internas associam um nome com uma cadeia de caracteres ou texto literal. A primeira entidade no Exemplo é uma entidade interna. Usando &ATI; em qualquer lugar do documento inserirá "SOAEXPERT, Inc" naquele local.

Entidades internas permitem a você definir atalhos para textos frequentemente digitados ou textos que se espera que sejam alterados, como o estado de revisão de um documento.

Entidades internas podem incluir referências para outras entidades internas, mas é errado elas serem recursivas.

A especificação XML pré-define cinco entidades internas:

- < produz o sinal de menor, <
- > produz o sinal de maior, >
- & produz o E comercial, &
- ' produz um apóstrofo, '
- " produz aspas, "

Entidades Externas

Entidades externas associam um nome com o conteúdo de um outro arquivo. Entidades externas permitem a documento XML referenciar o conteúdo de um outro arquivo; elas contêm texto ou dado binários. Se elas contêm texto, o conteúdo do arquivo externo é inserido no ponto de referência e analisado como parte do documento referente. Dados binários não são analisados e podem somente serem referenciados em um atributo; eles são usados para referenciar figuras e outro conteúdo não-XML no documento.

A segunda e a terceira entidades no Exemplo são entidades externas.

O uso de &boilerplate; inserirá o *conteúdo* do arquivo /standard/legalnotice.xml no local da referência da entidade. O processador XML analisará o conteúdo deste arquivo como se ele ocorresse literalmente no local.

A entidade **ATIlogo** também é uma entidade externa, mas o seu conteúdo é binário. A entidade **ATIlogo** pode ser usada somente como o valor de um atributo ENTITY (ou ENTITIES) (em um elemento **graphic**, talvez). O processador XML passará esta informação para a aplicação, mas ele não tenta processar o conteúdo de `/standard/logo.gif`.

Entidades Parâmetro

A entidade parâmetro somente pode ocorrer na declaração de tipo de documento. Uma declaração de uma entidade parâmetro é identificada por "% " (porcento e espaço) defronte ao seu nome na declaração. O sinal de porcento também é usado em referências para entidades parâmetro, ao invés do E comercial. As referências a entidade parâmetro são imediatamente expandidas na declaração de tipo de documento e seu texto de substituição é parte da declaração, onde as referências a entidades normais não são expandidas. Entidades parâmetro não são reconhecidas no corpo de um documento.

Voltando às declarações de elementos no Exemplo 2 , você perceberá que dois deles têm o mesmo modelo de conteúdo:

```
<!ELEMENT João (#PCDATA | citação)*>
<!ELEMENT José (#PCDATA | citação)*>
```

Até o momento, estes dois elementos são a mesma coisa somente porque eles têm a mesma definição literal. A fim de tornar mais explícito o fato de que estes dois elementos são semanticamente a mesma coisa, é usada uma entidade parâmetro para definir seus modelos de conteúdo. Há duas vantagens em se usar uma entidade parâmetro. Primeiramente, ela lhe permite dar um nome descriptivo ao conteúdo, e segundo que lhe permite alterar o modelo de conteúdo em somente um local, se você desejar atualizar as declarações do elemento, garantindo que elas sempre fiquem as mesmas:

```
<!ENTITY % pessoascontentes "#PCDATA | citação">
<!ELEMENT João (%pessoascontentes;)*>
<!ELEMENT José (%pessoascontentes;)*>
```

Declarações de Notação

Declarações de notação identificam tipos específicos de dados binários externos. Estas informações são passadas para a aplicação de processamento, que pode fazer o uso que quiser ou que desejar. Uma declaração de notação típica é:

```
<!NOTATION GIF87A SYSTEM "GIF">
```

Eu preciso de uma Declaração de Tipo de Documento?

Como foi visto, o conteúdo XML pode ser processado sem uma declaração de tipo de documento. Entretanto, existem alguns casos onde a declaração é necessária:

Ambientes de autoria

A maioria dos ambientes de autoria precisa ler e processar declarações de tipo de documento a fim de entender e reforçar o modelo de conteúdo do documento.

Valores padrões de atributos

Se um documento XML conta com valores padrões de atributos, pelo menos uma parte da declaração deve ser processada a fim de se obter os valores padrões corretos.

Manipulação de espaços em branco

A semântica associada com espaço em branco em conteúdo de elementos diferem da semântica associada com espaço em branco em conteúdo misturado. Sem um DTD, não há maneira para o processador distinguir os casos, e todos os elementos são efetivamente conteúdo misturado.

Incluindo uma Declaração de Tipo de Documento

Se presente, a declaração de tipo de documento deve ser a primeira coisa em um documento depois de comentários e instruções de processamento opcionais. A declaração de tipo de documento identifica o elemento raiz do documento e pode conter declarações adicionais. Todos os documentos XML devem ter um elemento raiz único que contenha todo o conteúdo do documento. Declarações adicionais podem vir de um DTD externo, chamado de subconjunto externo, ou ser incluído diretamente no documento, o subconjunto interno, ou ambos:

```

<?XML version="1.0" standalone="no"?>
<!DOCTYPE chapter SYSTEM "dbook.dtd" [
<!ENTITY %ulink.module "IGNORE">
<!ELEMENT ulink (#PCDATA)*>
<!ATTLIST ulink
    xml:link      CDATA  #FIXED "SIMPLE"
    xml-attributes CDATA  #FIXED "HREF URL"
    URL          CDATA  #REQUIRED> ]>
<chapter>...</chapter>

```

Este exemplo referencia um DTD externo, **dbook.dtd**, e inclui declarações de elementos e atributos para o elemento **ulink** no subconjunto interno. Neste caso, **ulink** dá a semântica de um link simples da especificação XLink.

Note que a declaração no subconjunto interno não leva em conta as declarações no subconjunto externo. O processador XML lê o subconjunto interno antes do externo e a *primeira* declaração tem precedência.

A fim de determinar se um documento é válido, o processador XML deve ler a declaração de tipo de documento inteira (ambos os subconjuntos). Mas para algumas aplicações, a validação pode não ser precisa, e pode ser suficiente para o processador ler somente o subconjunto interno. No exemplo acima, se a validade não é importante e a única razão para ler a declaração de tipo de documento é identificar a semântica de **ulink**, a leitura do subconjunto externo não é necessária.

Você pode comunicar estas informações na *declaração de documento standalone*. A declaração de documento **standalone**, **standalone="yes"** ou **standalone="no"**, ocorre na declaração XML. Um valor **yes** indica que somente declarações internas precisam ser processadas. Um valor **no** indica que *ambas* as declarações interna e externa devem ser processadas.

Outras questões de marcação

Além da marcação, existem algumas outras questões a considerar: manipulação de espaços em branco, normalização de valores dos atributos e a linguagem com a qual o documento foi escrito.

Manipulação de Espaços em Branco

A manipulação de espaços em brancos é uma questão sutil. Considere o seguinte fragmento de conteúdo:

```
<piada>
<João>Diga <citação>boa noite</citação>, Maria.</João>
```

O espaço em branco (a nova linha entre `<piada>` e `<João>`) é significante?

-Provavelmente não.

Mas como você pode afirmar isto? Você somente pode determinar se um espaço em branco é significante se você conhece o modelo de conteúdo dos elementos em questão. Em resumo, um espaço em branco é significante em conteúdo misturado e insignificante em conteúdo de elemento.

A regra para os processadores XML é que eles devem passar por todos os caracteres que não são marcação na aplicação. Se o processador é um processador de validação, ele também deve informar à aplicação se os caracteres espaços em branco são significantes.

O atributo especial `xml:space` pode ser usado para indicar explicitamente que os espaços em branco são significantes. Em qualquer elemento que inclua a especificação de atributo `xml:space='preserve'`, todos os espaços em branco naquele elemento (e dentro dos subelementos que não alteram explicitamente `xml:space`) serão significantes.

Os únicos valores válidos para `xml:space` são **preserve** e **default**. O valor **default** indica que o processamento padrão é desejado. Em um DTD, o atributo `xml:space` deve ser declarado como um tipo enumerado com somente estes dois valores.

Uma última observação sobre espaços em branco: em texto analisável, os processadores XML são requeridos para normalizar todas as marcas de final de linha para um único caracter de alimentação de linha (
). Isto raramente é de interesse dos autores, mas elimina um número de questões de portabilidade de plataformas cruzadas.

Caracteres especiais podem ser digitados usando referências de caracteres Unicode. Exemplo:

& = &.

Seções **CDATA** são formas alternativas de se usar dados de caracteres, como:

```
<![CDATA[<greeting>Hello, world!</greeting>]]>
```

Em um documento, uma seção **CDATA** instrui o analisador para ignorar a maioria dos caracteres de marcação.

Considere um código-fonte em um documento XML. Ele pode conter caracteres que o analisador XML iria normalmente reconhecer como marcação (< e &, por exemplo). Para prevenir isto, uma seção **CDATA** pode ser usada.

```
<![CDATA[ *p = &q; b = (i <= 3); ]]>
```

Entre o início da seção, **<![CDATA[**, e o fim da seção, **]]>**, todos os dados de caracteres são passados diretamente para a aplicação, sem interpretação. Elementos, referências a entidades, comentários e instruções de processamento são todos irreconhecíveis e os caracteres que os compõem são passados literalmente para a aplicação.

A única cadeia de caracteres que não pode ocorrer em uma seção **CDATA** é **"]]>"**.

Informações adicionais:

```
<!-- comment -->
```

Comentários iniciam com **<!--** e terminam com **-->**.

Os comentários podem conter qualquer dado, exceto a literal **--**. Você pode colocar comentários entre marcas em qualquer lugar em seu documento. Comentários não fazem parte de um conteúdo textual de um documento XML e é ignorado pelo processador.

```
<?target data...?>
```

Uma instrução para um processador; target identifica o processador para o qual ela foi direcionada e data é a string contendo a instrução.

9.0 XML Schemas – XSD

O que é XML Schemas ?

O XML Schema é uma alternativa à definição de tipos, DTD, e serve também para descrever uma estrutura de um documento XML.

O propósito de um *XML Schema* é definir os blocos de construção permitidos em um documento XML, como um **DTD**.

Um XML Schema:

- define elementos que podem aparecer em um documento
- define atributos que podem aparecer em um documento
- define que elementos são elementos filhos
- define a ordem dos elementos filhos
- define o número de elementos filhos
- define se um elemento é vazio ou pode incluir texto
- define tipos de dados para elementos e atributos
- define valores padrão e fixos para elementos e atributos

O elemento <schema>

```
<?xml version="1.0"?>                                <xs:schema>
...
...
</xs:schema>
```

O elemento `<schema>` pode conter alguns atributos. Uma declaração de Schema geralmente parece com isto:

```
<?xml version="1.0"?>                                <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.soaexpert.com.br"
xmlns="http://www.soaexpert.com.br"
elementFormDefault="qualified">
...
</xs:schema>
```

O seguinte fragmento:

```
<xs:schema  
xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

Indica que os elementos e tipos de dados usados no esquema (schema, element, complexType, sequence, string, boolean, etc.) vêm do namespace "<http://www.w3.org/2001/XMLSchema>". Ele também especifica que os elementos e tipos de dados que vêm de "<http://www.w3.org/2001/XMLSchema>" devem ser prefixados com xs.

Este fragmento:

```
targetNamespace="http://www.soaexpert.com.br"
```

Indica que os elementos definidos por este esquema (note, to, from, heading, body) vêm do namespace "<http://www.soaexpert.com.br>" .

Este fragmento:

```
xmlns="http://www.soaexpert.com.br"
```

indica que o namespace padrão é "<http://www.soaexpert.com.br>".

Este fragmento:

```
elementFormDefault="qualified">
```

Indica que todo elemento usado por uma instância de documento XML que foi declarado neste esquema deve ser qualificado pelo namespace.

9.1 Referenciando um Schema em um XML

Este documento XML tem uma referência para um XML Schema:

```
<?xml version="1.0"?>      <note
 xmlns="http://www.soaexpert.com.br"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.soaexpert.com.br note.xsd">
<to>Tove</to>    <from>Jani</from>
<heading>Reminder</heading>  <body>Don't forget me this
weekend!</body>  </note>
```

Este fragmento:

```
xmlns="http://www.soaexpert.com.br"
```

Especifica a declaração de namespace padrão. Esta declaração diz ao validador de esquema que os elementos usados neste documento XML são declarados no namespace

"<http://www.soaexpert.com.br>".

Uma vez que você tem uma instância XML Schema do namespace disponível:

```
xns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
```

Você pode usar o atributo schemaLocation. Este atributo tem dois valores. O primeiro é o namespace usado. O segundo é a localização do esquema XML para se usado pelo namespace:

```
xsi:schemaLocation="http://www.soaexpert.com.br
/note.xsd">
```

9.2 XSD Elemento simples

XML Schemas define os elementos de um arquivo XML.

Um elemento simples é um elemento XML que contém apenas texto. Ele não pode conter outros elementos ou atributos.

9.2.1 O que é um elemento simples

Um elemento simples é um elemento XML que contém apenas texto. Ele não pode conter outros elementos ou atributos.

Entretanto, a restrição "apenas texto" é pouco clara. O texto pode ser de diferentes tipos. Pode ser um dos tipos inclusos na definição de XML Schema (boolean, string, date, etc.), ou um tipo personalizado que você mesmo pode definir.

9.2.2 Como definir um elemento simples

A sintaxe para definir um elemento simples é:

```
<xs:element name="xxx" type="yyy"/>
```

onde xxx é o nome do elemento e yyy é o tipo de dado do elemento.

Aqui está alguns elementos XML:

```
<lastname>Refsnes</lastname> <age>34</age>
<dateborn>1968-03-27</dateborn>
```

E aqui as definições correspondentes:

```
<xs:element name="lastname" type="xs:string"/>
<xs:element name="age" type="xs:integer"/>      <xs:element
name="dateborn" type="xs:date"/>
```

9.2.3 Tipos de dados XML Schema comuns

XML Schema tem vários tipos de dados próprios. Aqui está uma lista dos mais comuns:

- xs:string
- xs:decimal
- xs:integer
- xs:boolean
- xs:date
- xs:time

9.2.4 Declare valores padrão e fixos para elementos simples

Elementos simples podem ter um conjunto de valores padrão OU um fixos.

Um valor padrão é automaticamente atribuído ao elemento quando nenhum outro valor é especificado. No exemplo seguinte, o valor padrão é "red":

```
<xs:element name="color" type="xs:string"
default="red"/>
```

Um valor fixo também é atribuído automaticamente ao elemento. Você não pode especificar outro valor. No exemplo a seguir, o valor fixo é "red":

```
<xs:element name="color" type="xs:string" fixed="red"/>
```

9.3 Atributos XSD

Todos atributos são declarados como tipos simples.

Apenas elementos complexos tem atributos !

9.3.1 O que é um atributo?

Elementos simples não podem ter atributos. Se um elemento tem atributos, ele é considerado do tipo complexo. Mas atributos são declarados como tipos simples. Isso significa que um elemento com atributos sempre têm uma definição do tipo complexo.

9.3.2 Como definir um atributo

A sintaxe para definir um atributo é:

```
<xs:attribute name="xxx" type="yyy"/>
```

onde xxx é o nome do atributo e yyy é o tipo de dado do atributo.

Aqui está um elemento XML com um atributo:

```
<lastname lang="EN">Smith</lastname>
```

E aqui a definição correspondente do atributo:

```
<xs:attribute name="lang" type="xs:string"/>
```

9.3.3 Tipos de dados comuns do XML Schema

XML Schema tem vários tipos de dados próprios. Aqui está uma lista dos mais comuns:

- xs:string

- xs:decimal
- xs:integer
- xs:boolean
- xs:date
- xs:time

9.3.4 Declare valores padrão e fixo para atributos

Atributos podem ter valores padrão ou fixo especificados.

Um valor padrão é atribuído automaticamente ao atributo quando nenhum outro valor é especificado. No exemplo seguinte, o valor padrão é "EN":

```
<xs:attribute name="lang" type="xs:string" default="EN">
```

Um valor fixo é atribuído automaticamente ao atributo. Você não pode especificar outro valor. No exemplo seguinte, o valor fixo é "EN":

```
<xs:attribute name="lang" type="xs:string" fixed="EN"/>
```

9.3.5 Criando atributos opcionais e obrigatórios

Todos atributos são opcionais por padrão. Para especificar explicitamente que um atributo é opcional, utilize o atributo "use":

```
<xs:attribute name="lang" type="xs:string"  
use="optional"/>
```

Para fazer um atributo obrigatório:

```
<xs:attribute name="lang" type="xs:string"  
use="required"/>
```

10.0 Restrições de Conteúdo

Quando um elemento ou um atributo XML tem um tipo definido, isto cria uma restrição ao conteúdo dele. Se um elemento XML é do tipo "xs:date" e contém um string como "Hello Mother", o elemento não vai ser validado.

Mas, há mais... com XML Schemas, você pode adicionar suas próprias restrições aos seus elementos e atributos XML. Estas restrições são chamadas *facets*. Você pode ler mais sobre *facets* no próximo capítulo.

10.1 Restrições/facets XSD

Restrições são usadas para controlar os valores aceitos para elementos e atributos XML. Restrições em elementos XML são achamados *facets*.

10.2 Restrições em valores

Este exemplo define um elemento chamado "age" com uma restrição. O valor de age não pode ser menor que 0 ou maior que 100:

```
<xs:element name="age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="100"/>
    </xs:restriction> </xs:simpleType> </xs:element>
```

10.3 Restrições

Para limitar o conteúdo de um elemento XML em um conjunto de valores aceitáveis, nós podemos usar restrições enumeradas. Este exemplo define um elemento chamado "car":

```
<xs:element name="car">    <xs:simpleType>
<xs:restriction base="xs:string">      <xs:enumeration
value="Audi"/>      <xs:enumeration value="Golf"/>
<xs:enumeration value="BMW"/>   </xs:restriction>
</xs:simpleType>  </xs:element>
```

O elemento "carro" é um tipo simples com restrição. Os valores aceitáveis são: Audi, Golf, BMW.

O exemplo acima também poderia ser escrito assim:

```
<xs:element name="car" type="carType">
<xs:simpleType name="carType">  <xs:restriction
base="xs:string">      <xs:enumeration value="Audi"/>
<xs:enumeration value="Golf"/>   <xs:enumeration
value="BMW"/>   </xs:restriction> </xs:simpleType>
```

Nota: Neste caso o tipo "carType" pode ser usado por outros elementos, porque ele não é parte do elemento "car".

10.4 Restrições Em Séries de Valores

Para limitar o conteúdo de um elemento XML em uma série de números ou letras, podemos utilizar a restrição de padrão.

Este exemplo define o elemento chamado "letter":

```
<xs:element name="letter">
<xs:simpleType>
  <xs:restriction base="xs:string">
    <xs:pattern value="[a-z]"/>
  </xs:restriction>
</xs:simpleType>
</xs:element>
```

O elemento "letter" é do tipo simples com uma restrição. O único valor aceitável é UMA das letras MINÚSCULAS de a até z.

O próximo exemplo define um elemento chamado "initials":

```
<xs:element name="initials">
    <xs:simpleType>
        <xs: restriction base="xs:string">
            <xs:pattern value="[A-Z][A-Z][A-Z]"/>
        <xs:restriction>
    </xs:simpleType> </xs:element>
```

O elemento "initials" é simples com uma restrição. O único valor aceitável são TRÊS letras MAIÚSCULAS de a até z.

Este exemplo também define um elemento chamado "initials":

```
<xs:element name="initials">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:patter value="[a-zA-Z][a-zA-Z][a-zA-Z]"/>
        </xs:restriction>
    <xs:simpleType>
    </xs:element>
```

O elemento "initials" é do tipo simples com restrição. O único valor aceitável são TRÊS letras MINÚSCULAS ou MAIÚSCULAS de a até z.

Este exemplo define um elemento chamado "choice":

```
<xs:element name="choice">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:pattern value="xyz"/>
        </xs:restriction>
    <xs:simpleType>
    </xs:element>
```

O elemento "choice" é um tipo simples com restrição. O único valor aceitável é UMA das seguintes letras: x, y, OU z.

O próximo exemplo define um elemento chamado "prodid":

```
<xs:element name="prodid">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:pattern value="[0-9][0-9][0-9][0-9][0-9]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

O elemento "prodid" é um tipo simples com restrição. O único valor aceitável são CINCO dígitos em seqüência, e cada dígito deve estar entre 0 e 9.

10.5 Outras restrições em séries de valores

Algumas outras restrições que podem ser definidas por restrição de padrão:

Este exemplo define um elemento chamado "letter":

```
<xs:element name="letter">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="([a-z])*/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

O elemento "letter" é um tipo simples com restrição. O valor aceitável é zero ou mais ocorrências de letras minúsculas de a até z.

Este exemplo também define um elemento chamado "letter":

```
<xs:element name="letter">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="([a-z][A-Z])+" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

O elemento "letter" é um tipo simples com uma restrição. O valor aceitável é uma ou mais ocorrências de uma letra minúscula seguida de uma letra maiúscula de a até z.

Este exemplo define um elemento chamado "gender":

```
<xs:element name="gender">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="male|female" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

O elemento "gender" é um tipo simples com uma restrição. O único valor aceitável é male OU female.

Este exemplo define um elemento chamado "password":

```
<xs:element name="password">
<xs:simpleType>
    <xs:restriction base="xs:string">
        <xs:pattern value="[a-zA-Z0-9]{8}"/>
    </xs:restriction>
</xs:simpleType>
<xs:element>
```

O elemento "password" é um tipo simples com um restrição. Deve haver exatamente oito caracteres e estes caracteres devem ser letras minúsculas ou maiúsculas de a até z, ou um número de 0 a 9.

10.6 Restrições em caracteres vazios

Para especificar como um caractere vazio deve ser tratado, devemos usar a restrição whiteSpace.

Este exemplo define um elemento chamado "address":

```
<xs:element name="address">
<xs:simpleType>
    <xs:restriction base="xs:string">
<xs:whiteSpace value="preserve"/>
    <xs:restriction>
</xs:simpleType>
</xs:element>
```

O elemento "address" é um tipo simples com uma restrição. A restrição whiteSpace é definida como "preserve", que significa que o processador XML NÂO VAI remover nenhum caractere vazio.

Este exemplo também define um elemento chamado "address":

```
<xs:element name="address">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:whiteSpace value="replace"/>
    </xs:restriction>
    </xs:simpleType>
</xs:element>
```

Este elemento "address" é um tipo simples com um restrição. A restrição whiteSpace é definida como "replace", que significa que o processador XML VAI SUBSTITUIR todos caracteres vazios (quebras de linha, tabs, espaços) com espaços.

Este exemplo também define um elemento chamado "address":

```
<xs:element name="address">
    <xs:simpleType>
        <xs:restriction base="xs:string">
            <xs:whiteSpace value="collapse"/>
    </xs:restriction>
    </xs:simpleType>
</xs:element>
```

Este elemento "address" é um tipo simples com uma restrição. A restrição whiteSpace é definida como "collapse", que significa que o processador XML VAI REMOVER todos caracteres vazios (quebras de linha, tabs, espaços são substituídos com espaços, espaços iniciais e finais são removidos, espaços múltiplos são reduzidos a um).

10.7 Restrições de comprimento

Para limitar o comprimento de um elemento, nós usamos as restrições de comprimento, maxLength e minLength.

Este exemplo define um elemento chamado "password":

```
<xs:element name="password">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:length value="8"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

O elemento "password" é um tipo simples com uma restrição. O valor deve ter exatamente oito caracteres.

Este exemplo define outro elemento chamado "password":

```
<xs:element name="password">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:minLength value="5"/>
      <xs:maxLength value="8"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Este elemento "password" é um tipo simples com uma restrição. O valor deve ter no mínimo cinco e no máximo oito caracteres.

10.8 Restrições para tipos de dados

Restrição	Descrição
enumeration	Define uma lista de valores válidos
fractionDigits	Especifica o número máximo casas decimais permitidas. Deve ser igual ou maior que zero
length	Especifica o número exato de caracteres ou itens permitidos. Deve ser igual ou maior que zero
maxExclusive	Especifica o valor máximo para valores numéricos (o valor deve ser menor que este valor)
maxInclusive	Especifica o valor máximo para valores numéricos (o valor deve ser menor ou igual a este valor)
maxLength	Especifica o número máximo de caracteres ou itens permitidos. Deve ser igual ou maior que zero
minExclusive	Especifica o valor mínimo para valores numéricos (o valor deve ser maior que este valor)
minInclusive	Especifica o valor mínimo para valores numéricos (o valor deve ser maior ou igual a este valor)
minLength	Especifica o número mínimo de caracteres ou itens permitidos. Deve ser igual ou maior que zero
pattern	Define a sequência exata de caracteres permitidos
totalDigits	Especifica o número exato de dígitos permitidos. Deve ser maior que zero
whiteSpace	Especifica como caracteres vazios (tabs, espaços e retornos de carro) são tratados

11.0 Elementos Complexos

Um elemento complexo contém outros elementos e/ou atributos.

O QUE É UM ELEMENTO COMPLEXO ?

Um elemento complexo é um elemento XML que contém outros elementos e/ou atributos.

Há quatro tipos de elementos complexos:

- elementos vazios
- elementos que contêm apenas outros elementos
- elementos que contêm apenas texto
- elementos que contêm tanto outros elementos quanto texto

Nota: Cada um desses elementos podem conter atributos também!

EXEMPLOS DE ELEMENTOS XML COMPLEXOS

Um elemento XML complexo, "product", que é vazio:

```
<product pid="1245"/>
```

Um elemento XML complexo, "employee", que contém apenas outros elementos:

```
<employee>
    <firstname>John</firstname>
    <lastname>Smith</lastname>
</employee>
```

Um elemento XML complexo, "food", que contém apenas texto:

```
<food type="dessert">Ice cream</food>
```

Um elemento XML complexo, "description", que contém outros elementos e texto:

```
<description> It happened on <date
lang="norwegian">03.03.99</date> .... </description>
```

COMO DEFINIR UM ELEMENTO COMPLEXO

Observe este elemento XML complexo, "employee", que contém apenas outros elementos:

```
<employee>
  <firstname>John</firstname>
  <lastname>Smith</lastname>
</employee>
```

Nós podemos definir um elemento complexo em um XML Schema de diferente maneiras:

1. O elemento "employee" pode ser declarado diretamente, nomeando o elemento:

```
<xsd:element name="employee">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="firstname" type="xsd:string"/>
      <xsd:element name="lastname" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Se você usar o método descrito acima, apenas o elemento "**employee**" pode usar o tipo complexo definido. Note que os elementos filhos, "firstname" e "lastname", são envolvidos pelo indicador **<sequence>**. Isto significa que os elementos filhos devem aparecer na mesma ordem da declaração: "firstname" primeiro e "lastname" depois. Você vai aprender sobre indicadores no capítulo Indicadores XSD.

2. O elemento "employee" pode ter um atributo tipo que faz referência ao nome do tipo complexo que deve ser usado:

```
<xs:element name="employee" type="personinfo"/>
<xs:complexType name="personinfo"> <xs:sequence>
<xs:element name="firstname" type="xs:string"/>
<xs:element name="lastname" type="xs:string"/>
</xs:sequence> .....
```

Se você usar o método descrito acima, vários elementos podem fazer referência ao mesmo tipo complexo, assim:

```
<xs:element name="employee" type="personinfo"/>
<xs:element name="student" type="personinfo"/>
<xs:element name="member" type="personinfo"/>
<xs:complexType name="personinfo"> <xs:sequence>
<xs:element name="firstname" type="xs:string"/>
<xs:element name="lastname" type="xs:string"/>
</xs:sequence> </xs:complexType>
```

Você também pode basear um elemento do tipo complexo em um tipo complexo existente e adicionar alguns elementos, assim:

```
<xs:element name="employee" type="fullpersoninfo"/>
<xs:complexType name="personinfo"> <xs:sequence>
<xs:element name="firstname" type="xs:string"/>
<xs:element name="personinfo" type="xs:string"/>
</xs:sequence> </xs:complexType> <xs:complexType
name="fullpersoninfo"> <xs:complexContent>
<xs:extension base="personinfo"> <xs:sequence>
<xs:element name="address" type="xs:string"/>
<xs:element name="city" type="xs:string"/>
<xs:element name="country" type="xs:string"/>
</xs:sequence> </xs:extension>
</xs:complexContent> </xs:complexType>
```

11.1 Elementos Complexos Vazios

Um elemento complexo vazio pode conter atributos; mas não pode ter nenhum conteúdo entre as tags de abertura e fechamento.

DEFINA TIPOS COMPLEXOS PARA ELEMENTOS VAZIOS

Um elemento XML vazio:

```
<product prodid="1345"/>
```

O elemento "product" acima não tem conteúdo. Para definir um tipo sem conteúdo, nós devemos definir um tipo que permita apenas elementos em seu conteúdo, mas nós não declaramos nenhum elemento realmente, assim:

```
<xss:element name="product">
    <xss:complexType>
        <xss:complexContent>
            <xss:restriction base="xss:integer">
                <xss:attribute name="prodid" type="xss:positiveInteger"/>
            </xss:restriction>
        </xss:complexContent>
    </xss:complexType>
</xss:element>
```

No exemplo acima, definimos um `complexType` tento `complexContent`, i.e. apenas elementos. O elemento `complexContent` diz que queremos restringir ou extender o modelo de conteúdo de um tipo complexo, e a restrição de inteiros declara um atributo, mas não cria nenhum elemento.

Entretanto, é possível declarar o elemento `product` de forma mais compacta, assim:

```
<xs:element name="product">
    <xs:complexType>
        <xs:attribute name="prodid"
type="xs:positiveInteger"/>
    </xs:complexType>
</xs:element>
```

Ou você pode dar um nome ao `complexType`, e fazer o elemento `"product"` ter um tipo que faz referência ao nome do `complexType` (se você usar este método, vários elementos podem referenciar o mesmo tipo complexo):

```
<xs:element name="product" type="prodtype"/>

<xs:complexType name="prodtype">
    <xs:attribute name="prodid"
type="xs:positiveInteger"/>
</xs:complexType>
```

11.2 Tipos complexos apenas elementos

Um tipo complexo "apenas elementos" contém um elemento que contém apenas outros elementos.

DEFINA TIPOS COMPLEXOS APENAS COM ELEMENTOS

Um elemento XML, `"person"`, que contém apenas outros elementos:

```
<person> <firstname>John</firstname>
<lastname>Smith</lastname> </person>
```

Você pode definir o elemento "person" em um esquema, assim:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Note a tag `<xs:sequence>`. Isto significa que os elementos definidos ("firstname" e "lastname") devem aparecer nesta ordem dentro do elemento "person".

Ou você pode dar um nome ao `complexType`, e um atributo `type` ao elemento "person" que faz referência a ele (se você usar este método, vários elementos podem referenciar o mesmo tipo complexo).

```
<xs:element name="person" type="persontype"/>

<xs:complexType name="persontype">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

11.3 Elementos Complexos apenas textos

Um elemento complexo texto pode conter tanto atributos quanto texto.

DEFINA UM ELEMENTO COMPLEXO APENAS TEXTO

Este tipo contém apenas conteúdo simples (texto e atributos), assim nós adicionamos um elemento simpleContent em torno do conteúdo. Quando usar conteúdo simples, você precisa definir uma extensão ou uma restrição com o elemento simpleContent, assim:

```
<xs:element name="somename">      <xs:complexType>
<xs:simpleContent>          <xs:extension base="basetype">
    ...
    </xs:simpleContent>  </xs:complexType> </xs:element>
OU
<xs:complexType>          <xs:element name="somename">
    <xs:simpleContent>
        ...
        </xs:simpleContent>
    </xs:complexType> </xs:element>
```

Dica: Use o elemento extension para expandir o tipo simples base de um elemento, e o elemento restriction para limitá-lo.

Aqui está um exemplo de um elemento XML, "shoesize", que contém apenas texto:

```
<shoesize country="france">35</shoesize>
```

O exemplo a seguir declara um complexType, "shoesize". O conteúdo é definido como tipo de dado integer e o elemento "shoesize" também contém um atributo chamado "country":

```
<xs:element name="shoesize">      <xs:complexType>
<xs:simpleContent>          <xs:extension base="xs:integer">
    <xs:attribute name="country" type="xs:string"/>
</xs:extension>          </xs:simpleContent>
</xs:complexType> </xs:element>
```

Também poderíamos definir um nome para o complexType, e especificar um atributo type ao elemento "shoesize" que faz referência ao nome do complexType (se você usar este método, vários elementos podem referenciar o mesmo tipo complexo):

```
<xs:element name="shoesize" type="shoetype"/>
<xs:complexType name="shoetype"> <xs:simpleContent>
<xs:extension base="xs:integer"> <xs:attribute
name="country" type="xs:string"/> </xs:extension>
</xs:simpleContent> </xs:complexType>
```

11.4 Elementos Complexos Mistas

Um elemento do tipo complexo misto pode conter atributos, elementos, e texto.

DEFINA TIPOS COMPLEXOS COM CONTEÚDO MISTO

Um elemento XML, "letter", que contém tanto outros elementos quanto texto:

```
<letter> Dear Mr.<name>John Smith</name> Your order
<orderid>1032</orderid> wil be shipped on
<shipdate>2001-07-13</shipdate>. </letter>
```

Note o texto que aparece entre os elementos "name", "orderid", e "shipdate" são todos filhos de "letter". O seguinte esquema declara o elemento "letter":

```
<xs:element name="letter"> <xs:complexType
mixed="true"> <xs:sequence> <xs:element
name="name" type="xs:string"/> <xs:element
name="orderid" type="xs:positiveInteger"/>
<xs:element name="shipdate" type="xs:date"/>
</xs:sequence> </xs:complexType> <xs:element>
```

Nota: Para permitir que caracteres apareçam entre os elementos filhos de "letter", o atributo mixed deve ser definido como "true". A tag <xs:sequence> significa que os elementos definidos (name, orderid e shipdate) devem parecer nesta ordem no elemento "letter".

Nós também poderíamos dar um nome ao elemento complexType, e definir o atributo type de "letter" como uma referência a ele (se você usar este método, vários elementos podem fazer referência ao mesmo complexType):

```
<xs:element      name="letter"      type="lettertype"/>
<xs:complexType   name="lettertype"   mixed="true">
<xs:sequence>           <xs:element name="name"
type="xs:string"/>           <xs:element name="orderid"
type="xs:positiveInteger"/>           <xs:element
name="shipdate" type="xs:date"/>           </xs:sequence>
</xs:complexType>
```

11.5 Indicadores de tipos Complexos

Nós podemos controlar COMO os elementos serão usados em documentos através de indicadores.

INDICADORES

Nós temos sete tipos de indicadores:

Indicadores de ordem:

- All
- Choice
- Sequence

Indicadores de ocorrência:

- maxOccurs
- minOccurs

Indicadores de grupo:

- Group name
- attributeGroup name

INDICADORES DE ORDEM

Indicadores de ordem são usados para definir a ordem em que os elementos ocorrem.

INDICADOR ALL

O indicador `<all>` especifica por padrão que os elementos filhos podem aparecer em qualquer ordem e que cada um deve ocorrer, e apenas uma vez:

```
<xs:element name="person">          <xs:complexType>
<xs:all>                            <xs:element name="firstname"
                                         <xs:element name="lastname"
                                         </xs:all>      </xs:complexType>
                                         type="xs:string"/>
                                         type="xs:string"/>
</xs:element>
```

Nota: Usando o indicador `<all>` você pode especificar o indicador `<minOccurs>` em 0 ou 1 e `<maxOccurs>` só pode ser 1 (`<minOccurs>` e `<maxOccurs>` são descritos adiante).

INDICADOR CHOICE

O indicador `<choice>` especifica que um elemento filho ou outro pode ocorrer:

```
<xs:element name="person">          <xs:complexType>
<xs:choice>                          <xs:element name="employee"
                                         <xs:element name="member"
                                         </xs:choice>    </xs:complexType>
                                         type="employee"/>
                                         type="member"/>
</xs:element>
```

INDICADOR SEQUENCE

O indicador <sequence> especifica que os elementos filhos devem aparecer em uma ordem específica:

```
<xs:element name="person">          <xs:complexType>
<xs:sequence>                      <xs:element name="firstname"
type="xs:string"/>                  <xs:element name="lastname"
type="xs:string"/>                  </xs:sequence>
</xs:complexType> </xs:element>
```

INDICADOR DE OCORRÊNCIA

Indicadores de ocorrência são usados para indicar com que freqüência um elemento pode ocorrer.

Nota: Para todos indicadores "Order" e "Group" (any, all, choice, sequence, group, name, e group reference) o valor padrão de maxOccurs e minOccurs é 1!!!

INDICADOR MAXOCCURS

O indicador <maxOccurs> especifica o número máximo de vezes que um elemento pode ocorrer:

```
<xs:element name="person">          <xs:complexType>
<xs:sequence>                      <xs:element name="full_name"
type="xs:string"/>                  <xs:element name="child_name"
name="child_name" type="xs:string"  maxOccurs="10"/>
</xs:sequence> </xs:complexType> </xs:element>
```

O exemplo acima indica que o elemento "child_name" pode ocorre no mínimo uma vez (o valor padrão para minOccurs é 1) e no máximo dez vezes em um elemento "person".

INDICADOR MINOCCURS

O indicador <minOccurs> especifica o número mínimo de vezes que um elemento pode ocorrer:

```
<xs:element name="person">          <xs:complexType>
<xs:sequence>                      <xs:element name="full_name"
type="xs:string"/>                  <xs:element name="child_name"
type="xs:string"                   maxOccurs="10" minOccurs="0"/>
</xs:sequence>    </xs:complexType> </xs:element>
```

O exemplo acima indica que o elemento "child_name" pode ocorrer um mínimo de zero vezes e um máximo de dez em um elemento "person".

Para permitir que um elemento pareça um número ilimitado de vezes, use a instrução maxOccurs="unbounded":

UM EXEMPLO PRÁTICO

Um arquivo XML chamado "Myfamily.xml":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<persons
  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="family.xsd">
  <person>          <full_name>Hege Refsnes</full_name>
  <child_name>Cecilie</child_name> </person> <person>
  <full_name>Tove           Refsnes</full_name>
  <child_name>Hege</child_name>
  <child_name>Stale</child_name>
  <child_name>Jim</child_name>
  <child_name>Borge</child_name> </person> <person>
  <full_name>Stale   Refsnes</full_name>      </person>
</persons>
```

O arquivo XML acima contém um elemento raiz chamado "persons". Dentro deste elemento está definido vários elementos "person". Cada elemento "person" deve conter um elemento filho "full_name" e pode conter até cinco elementos filho "child_name".

Aqui está o esquema "family.xsd":

```
<xml version="1.0" encoding="ISO-8859-1"?>
<xsschema
  xmlns:xss="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
  <xsschema>
    <xselement name="persons">
      <xssimpleType>
        <xsssequence>
          <xselement name="person" maxOccurs="unbounded">
            <xssimpleType>
              <xselement name="full_name" type="xs:string"/>
              <xselement name="child_name" type="xs:string"
                minOccurs="0" maxOccurs="5"/>
            </xssimpleType>
          </xsssequence>
        </xssimpleType>
      </xselement>
    </xsschema>
  </xsschema>
</xsschema>
```

INDICADORES DE GRUPO

Indicadores de grupo são usados para definir grupos de elementos relacionados.

ELEMENTOS DE GRUPO

Elementos de grupo são definidos com a declaração group, assim:

```
<xsgroup name="groupname"> ... </xsgroup>
```

Você deve definir um elemento all, choice, ou sequence dentro da declaração group. O exemplo seguinte define um grupo chamado "persongroup", que define um grupo de elementos que devem ocorrer em uma sequência exata:

```

<xs:group name="persongroup">
<xs:element name="firstname" type="xs:string"/>
<xs:element name="lastname" type="xs:string"/>
<xs:element name="birthday" type="xs:date"/>
</xs:sequence> </xs:group>

```

Depois que você definiu um grupo, você pode referenciá-lo na definição de outro grupo ou tipo complexo, assim:

```

<xs:group name="persongroup">
<xs:element name="firstname" type="xs:string"/>
<xs:element name="lastname" type="xs:string"/>
<xs:element name="birthday" type="xs:date"/>
</xs:sequence> </xs:group>
<xs:complexType name="person" type="personinfo"/>
<xs:sequence> <xs:group name="personinfo">
<xs:element name="country" type="xs:string"/> </xs:sequence> </xs:group>
<xs:element name="persongroup" ref="persongroup"/>
</xs:sequence> </xs:complexType>

```

GRUPOS DE ATRIBUTOS

Grupos de atributos são definidos com a declaração attributeGroup, assim:

```

<xs:attributeGroup name="groupname"> ...
</xs:attributeGroup>

```

O exemplo a seguir define um grupo de atributos chamado "personattrgroup":

```

<xs:attributeGroup name="personattrgroup">
<xs:attribute name="firstname" type="xs:string"/>
<xs:attribute name="lastname" type="xs:string"/>
<xs:attribute name="birthday" type="xs:date"/>
</xs:attributeGroup>

```

Depois que você definiu um grupo de atributos, você pode referenciá-lo na definição de outro grupo ou tipo complexo, assim:

```
<xs:attributeGroup name="personattrgroup">
<xs:attribute name="firstname" type="xs:string"/>
<xs:attribute name="lastname" type="xs:string"/>
<xs:attribute name="birthday" type="xs:date"/>
</xs:attributeGroup>
<xs:element name="person" ref="personattrgroup"/>
</xs:element>
```

12.0 O Elemento <ANY>

O elemento <any> nos permite extender o documento XML com elementos não especificados no esquema.

O exemplo a seguir é um fragmento de um esquema XML chamado "family.xsd". Ele mostra uma declaração para o elemento "person". Usando o elemento <any> nós podemos extender (depois de <lastname>) o conteúdo de "person" com qualquer elemento:

```
<xs:element name="person"> <xs:complexType>
<xs:sequence> <xs:element name="firstname"
type="xs:string"/> <xs:element name="lastname"
type="xs:string"/> <xs:any minOccurs="0"/>
</xs:sequence> </xs:complexType> </xs:element>
```

Agora nós queremos extender o elemento "person" com um elemento "children". Neste caso nós podemos fazer isto, mesmo que o autor do esquema acima nunca tenha declarado o elemento "children"!

Observer esse arquivo de esquema, chamado "children.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.soaexpert.com.br"
  xmlns="http://www.soaexpert.com.br"
  elementFormDefault="qualified">
  <xsd:element
    name="children">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element
          name="childname"
          type="xsd:string"
          maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

O arquivo XML abaixo (chamado "Myfamily.xml"), usa componentes de dois esquemas diferentes; "family.xsd" e "children.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1"?> <person
  xmlns="http://www.microsoft.com"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance" xsi:SchemaLocation="http://www.microsoft.com
  family.xsd http://www.soaexpert.com.br children.xsd">
  <person>
    <firstname>Hege</firstname>
    <lastname>Refsnes</lastname>
    <children>
      <childname>Cecilie</childname>
    </children>
  </person>
  <person>
    <firstname>Stale</firstname>
    <lastname>Refsnes</lastname>
  </person>
</persons>
```

O arquivo XML acima é válido porque o esquema "family.xsd" nos permite extender o elemento "person" com um elemento opcional depois do elemento "lastname"!

Os elementos `<any>` e `<anyAttribute>` são usados para fazer documentos EXTENSÍVEIS! Eles permitem aos documentos conterem elementos adicionais que não estão declarados no esquema XML!

12.1 O Elemento `<ANYATTRIBUTE>`

O elemento `<anyAttribute>` nos permite extender o documento XML com atributos que não foram especificados no esquema!

O ELEMENTO <ANYATTRIBUTE>

O elemento <anyAttribute> nos permite extender o documento XML com atributos que não foram especificados no esquema.

O exemplo a seguir é um fragmento de um esquema XML chamado "family.xsd". Ele mostra uma declaração para o elemento "person". Usando o elemento <anyAttribute> nós podemos adicionar qualquer número de atributos ao elemento "person":

```
<xs:element name="person">          <xs:complexType>
<xs:sequence>                      <xs:element name="firstname"
type="xs:string"/>                  <xs:element name="lastname"
type="xs:string"/>                  </xs:sequence>
<xs:anyAttribute/>    </xs:complexType> </xs:element>
```

Agora nós queremos extender o elemento "person" com um atributo "gender". Neste caso nós podemos fazer isto, mesmo que o autor do esquema acima nunca tenha declarado nenhum atributo "gender"!

Observe este arquivo de esquema, chamado "attribute.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://w3schools.com"
  xmlns="http://www.soaexpert.com.br"
  elementFormDefault="qualified">           <xs:attribute
  name="gender">    <xs:simpleType>        <xs:restriction
  base="xs:string">      <xs:pattern value="male|female"/>
  </xs:restriction>    </xs:simpleType>   </xs:attribute>
</xs:schema>
```

O arquivo XML abaixo (chamado "Myfamily.xml"), utiliza componentes de dois esquemas diferentes; "family.xsd" e "attribute.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<persons xmlns="http://www.microsoft.com"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance" xsi:SchemaLocation="http://www.microsoft.com
  family.xsd http://www.soaexpert.com.br attribute.xsd">
  <person gender="female"> <firstname>Hege</firstname>
  <lastname>Refsnes</lastname> </person> <person
  gender="male"> <firstname>Stale</firstname>
  <lastname>Refsnes</lastname> </person> </persons>
```

O arquivo XML acima é válido porque o esquema "family.xsd" nos permite adicionar um atributo ao elemento "person"!

Os elemento `<any>` e `<anyAttribute>` são usado para fazer documentos EXTENSÍVEIS! Eles permitem aos documentos conterem elementos adicionais que não foram declarados no esquema XML!

12.2 Substituição de Elementos

Com XML Schemas um elemento pode substituir outro.

SUBSTITUIÇÃO DE ELEMENTOS

Digamos que nós temos usuários de dois países diferentes: Inglaterra e Noruega. Nós gostaríamos de permitir ao usuário escolher se quer usar os nomes de elementos em inglês ou norueguês no documento XML.

Para resolver este problema, nós podemos definir um **substitutionGroup** no esquema XML. Primeiro nós declaramos um elemento chave e então nós declaramos outros elementos que são substituições para o elemento chave:

```
<xss:element name="name" type="xs:string"/>
<xss:element name="navn" substitutionGroup="name"/>
```

No exemplo acima, o elemento "name" é o elemento chave e "navn" o elemento substituível para "name".

Observe esse fragmento de um esquema XML:

```
<xs:element name="name" type="xs:string"/>
<xs:element name="navn" substitutionGroup="name"/>
<xs:complexType name="custinfo"> <xs:sequence>
<xs:element ref="name"/> </xs:sequence>
</xs:complexType> <xs:element name="customer"
type="custinfo"/> <xs:element name="kunde"
substitutionGroup="customer"/>
```

Um documento XML válido (de acordo com o esquema acima) poderia parecer com isso:

```
<customer> <name>John Smith</name> </customer>
```

ou com isso:

```
<kunde> <navn>John Smith</navn> </kunde>
```

BLOQUEANDO SUBSTITUIÇÃO DE ELEMENTOS

Para prevenir que outros elementos substituam um elemento específico, utilize o atributo `block`:

```
<xs:element name="name" type="xs:string"
block="substitution"/>
```

Observe esse fragmento de um esquema XML:

```
<xs:element name="name" type="xs:string"
block="substitution"/> <xs:element name="navn"
substitutionGroup="name"/> <xs:complexType
name="custinfo"> <xs:sequence> <xs:element
ref="name"/> </xs:sequence> </xs:complexType>
<xs:element name="customer" type="custinfo"
block="substitution"/> <xs:element name="kunde"
substitutionGroup="customer"/>
```

Um documento XML válido (de acordo com o esquema acima) parece-se com isso:

```
<customer> <name>John Smith</name> </customer>
```

MAS ISSO NÃO É MAIS VÁLIDO:

```
<kunde> <navn>John Smith</navn> </kunde>
```

UTILIZANDO SUBSTITUTIONGROUP

O tipo do elemento de substituição deve ser o mesmo, ou derivado, do tipo do elemento chave. Se o tipo do elemento de substituição é o mesmo do elemento chave você não precisará especificá-lo.

Note que todos os elementos em um substitutionGroup (o elemento chave e os de substituição) devem ser declarados como elementos globais, caso contrário não funcionará!

O QUE SÃO ELEMENTOS GLOBAIS?

Elementos globais são aqueles que são filhos imediatos do elemento "schema"! Elementos locais são elementos aninhados a outros elementos!

12.3 Exemplo

Neste capítulo vai demonstrar como escrever um XML Schema. Você também aprenderá que um esquema pode ser escrito de diferentes maneiras.

UM DOCUMENTO XML

Vamos dar uma olhada nesse documento XML chamado "shiporder.xml":

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<shiporder orderid="889923"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="shiporder.xsd">
  <orderperson>John Smith</orderperson> <shipto>
    <name>Ola Nordmann</name> <address>Langgt
    23</address> <city>4000 Stavanger</city>
    <country>Norway</country> </shipto> <item>
      <title>Empire Burlesque</title> <note>Special
      Edition</note> <quantity>1</quantity>
      <price>10.90</price> </item> <item> <title>Hide your
      heart</title> <quantity>1</quantity>
      <price>9.90</price> </item>
</shiporder>
```

documento XML acima consiste de um elemento raiz, "shiporder", que contém um atributo obrigatório chamado "orderid". O elemento "shiporder" contém três elementos filhos diferentes: "orderperson", "shipto" e "item". O elemento "item" aparece duas vezes e contém um elemento "title", um elemento opcional "note", um elemento "quantity" e um "price".

A linha acima:

`xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`
diz ao parser XML que o documento deve ser validado por um esquema.

A linha `xsi: noNamespaceSchemaLocation="shiporder.xsd"` especifica ONDE o esquema reside (no caso, ele está no mesmo diretório que "shiporder.xml").

CRIE UM XML SCHEMA

Agora vamos criar um esquema para o documento XML acima!

Começamos abrindo um novo arquivo que chamaremos "shiporder.xsd". Para criar o esquema podemos simplesmente seguir a estrutura no documento XML e definir cada elemento a medida que o encontramos. Vamos começar com a declaração XML padrão, seguida do elemento xs:schema que define um esquema:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  ...
  </xs:schema>
```

No esquema acima usamos o namespace padrão (xs), e a URI associada com este namespace na linguagem de definição Schema, que tem o valor padrão <http://www.w3.org/2001/XMLSchema>.

Em seguida, temos que definir o elemento "shiporder". Este elemento tem um atributo e contém outros elementos, assim o consideramos um tipo complexo. Os elementos filhos de "shiporder" são englobados por um elemento xs:sequence que define uma ordem para eles:

```
<xs:element name="shiporder">           <xs:complexType>
  <xs:sequence>          ...          ...
    </xs:sequence>
  </xs:complexType> </xs:element>
```

Então temos que definir o elemento "orderperson" como um tipo simples (porque ele não contém nenhum atributo ou elemento). O tipo (xs:string) é prefixado com o prefixo do namespace associado ao XML Schema que indica um tipo de dado pré-definido:

```
<xs:element name="orderperson" type="xs:string"/>
```

Em seguida temos que definir dois elementos que são do tipo complexo: "shipto" e "item". Começamos definindo o elemento "shipto":

```

<xs:element name="shipto">           <xs:complexType>
<xs:sequence>                         <xs:element name="name"
type="xs:string"/>                   <xs:element name="address"
type="xs:string"/>                   <xs:element name="city"
type="xs:string"/>                   <xs:element name="country"
type="xs:string"/>                   </xs:sequence>
</xs:complexType> </xs:element>

```

Com esquemas podemos definir o número possível de ocorrências de um elemento com os atributos maxOccurs e minOccurs. maxOccurs especifica o número máximo de ocorrências para um elemento e minOccurs o número mínimo. O valor padrão para maxOccurs e minOccurs é 1!!!

Agora podemos definir o elemento "item". Este elemento pode aparecer muitas vezes dentro do elemento "shiporder". Isto é especificado definindo o atributo maxOccurs do elemento "item" para "unbounded" que significa que podem haver tantas ocorrências do elemento "item" quanto o autor desejar. Note que o elemento "note" é opcional. Nós especificamos isto definindo o atributo minOccurs em zero:

```

<xs:element name="item" maxOccurs="unbounded">
<xs:complexType> <xs:sequence> <xs:element
name="title" type="xs:string"/> <xs:element
name="note" type="xs:string" minOccurs="0"/>
<xs:element name="quantity" type="xs:positiveInteger"/>
<xs:element name="price" type="xs:decimal"/>
</xs:sequence> </xs:complexType> </xs:element>

```

Agora podemos declarar o atributo do elemento "shiporder". Como ele é um atributo obrigatório nós especificamos e="required".

Nota: As declarações de atributos sempre devem vir no final:

```
<xs:attribute name="orderid" type="xs:string"  
use="required"/>
```

Aqui está a listagem completa do arquivo schema chamado "shiporder.xsd":

```
<?xml version="1.0" encoding="ISO-8859-1" ?>  
<xs:schema  
xmlns:xs="http://www.w3.org/2001/XMLSchema">  
<xs:element name="shiporder"> <xs:complexType>  
<xs:sequence> <xs:element name="orderperson"  
type="xs:string"/> <xs:element name="shipto">  
<xs:complexType> <xs:sequence>  
<xs:element name="name" type="xs:string"/>  
<xs:element name="address" type="xs:string"/>  
<xs:element name="city" type="xs:string"/>  
<xs:element name="country" type="xs:string"/>  
</xs:sequence> </xs:complexType>  
</xs:element> <xs:element name="item"  
maxOccurs="unbounded"> <xs:complexType>  
<xs:sequence> <xs:element name="title"  
type="xs:string"/> <xs:element name="note"  
type="xs:string" minOccurs="0"/> <xs:element  
name="quantity" type="xs:positiveInteger"/>  
<xs:element name="price" type="xs:decimal"/>  
</xs:sequence> </xs:complexType>  
</xs:element> </xs:sequence> <xs:attribute  
name="orderid" type="xs:string" use="required"/>  
</xs:complexType> </xs:element> </xs:schema>
```

DIVIDA O SCHEMA

O método de design anterior é muito simples, mas pode ser muito difícil de ler e manter quando os documentos são complexos!

O próximo método de design é baseado na definição de todos os elementos e atributos primeiro, e então referenciá-los usando o atributo ref.

Aqui está a nova aparência o arquivo de esquema ("shiporder.xsd"):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsschema
  xmlns:xss="http://www.w3.org/2001/XMLSchema">      <!--
  definição de elementos simples -->      <xss:element
  name="orderperson" type="xss:string"/>      <xss:element
  name="name" type="xss:string"/>      <xss:element
  name="address" type="xss:string"/>      <xss:element
  name="city" type="xss:string"/>      <xss:element
  name="country" type="xss:string"/>      <xss:element
  name="title" type="xss:string"/>      <xss:element name="note"
  type="xss:string"/>      <xss:element name="quantity"
  type="xss:positiveInteger"/>      <xss:element name="price"
  type="xss:decimal"/>      <!-- definição de atributos -->
  <xss:attribute name="orderide" type="xss:string"/>      <!--
  definição de elementos complexos -->      <xss:element
  name="shipto">      <xss:complexType>      <xss:sequence>
  <xss:element ref="name"/>      <xss:element
  ref="address"/>      <xss:element ref="city"/>
  <xss:element ref="country"/>      </xss:sequence>
  </xss:complexType>      </xss:element>      <xss:element
  name="item">      <xss:complexType>      <xss:sequence>
  <xss:element ref="title"/>      <xss:element ref="note"
  minOccurs="0"/>      <xss:element ref="quantity"/>
  <xss:element ref="price"/>      </xss:sequence>
  </xss:complexType>      </xss:element>      <xss:element
  name="shiporder">      <xss:complexType>
  <xss:sequence>      <xss:element ref="orderperson"/>
  <xss:element ref="shipto"/>      <xss:element ref="item"
  maxOccurs="unbounded"/>      </xss:sequence>
  <xss:attribute ref="orderid" use="required"/>
  </xss:complexType> </xss:element> </xss:schema>
```

USANDO TIPOS NOMEADOS

O terceiro método de design define classes e tipos, que nos permite reutilizar a definição de elementos. Isto é feito nomeando os elementos simpleTypes e complexTypes, e então apontando para eles com o atributo type do elemento.

Aqui está a nova forma do arquivo de esquema ("shiporder.xsd"):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:simpleType name="stringtype"> <xs:restriction
    base="xs:string"/> </xs:simpleType> <xs:simpleType
    name="inttype"> <xs:restriction
    base="xs:positiveInteger"/> </xs:simpleType>
    <xs:simpleType name="dectype"> <xs:restriction
      base="xs:decimal"/> </xs:simpleType> <xs:simpleType
      name="orderidtype"> <xs:restriction base="xs:string">
        <xs:pattern value="[0-9]{6}" /> </xs:restriction>
      </xs:simpleType> <xs:complexType name="shiptotype">
        <xs:sequence> <xs:element name="name"
          type="stringtype"/> <xs:element name="address"
          type="stringtype"/> <xs:element name="city"
          type="stringtype"/> <xs:element name="country"
          type="stringtype"/> </xs:sequence> </xs:complexType>
        <xs:complexType name="itemtype"> <xs:sequence>
          <xs:element name="title" type="stringtype"/>
          <xs:element name="note" type="stringtype"/>
          <xs:element name="quantity" type="inttype"/>
          <xs:element name="price" type="dectype"/>
        </xs:sequence> </xs:complexType> <xs:complexType
        name="shipordertype"> <xs:sequence>
          <xs:element name="orderperson" type="stringtype"/>
          <xs:element name="shipto" type="shiptotype"/>
          <xs:element name="item" maxOccurs="unbounded" type="itemtype"/>
        </xs:sequence> <xs:attribute name="orderid"
        type="orderidtype" use="required"/> </xs:complexType>
      <xs:element name="shiporder" type="shipordertype"/>
    </xs:schema>
```

O elemento restriction indica que o tipo de dado é derivado de um namespace de XML Schema do W3C. Assim, esse fragmento:

```
<xs:restriction base="xs:string"/>
```

significa que o valor do elemento ou atributo de ser uma string.

O elemento restriction costuma ser mais usado para aplicar restrições nos elementos. Observe as seguintes linhas do esquema acima:

```
<xs:simpleType name="orderidtype">      <xs:restriction  
base="xs:string">                  <xs:pattern value="[0-9]{6}" />  
</xs:restriction> </xs:simpleType>
```

Isto indica que o valor do elemento ou atributo deve ser uma string e deve exatamente seis caracteres e cada caracter deve ser um número entre 0 e 9.

TIPOS DE DADOS DE STRING

Tipos de dados string são usados para valores que contém cadeias dcaracteres.

TIPO DE DADO STRING

O tipo de dado string pode conter caracteres, quebras de linha, retornos de carro, e caracteres de tabulação.

A seguir está o exemplo de uma declaração de string em um esquema:

```
<xs:element name="customer" type="xs:string"/>
```

Um elemento em seu documento poderia ser assim:

```
<customer>John Smith</customer>
```

Ou assim:

```
<customer> John Smith </customer>
```

Nota: O processador XML não vai modificar o valor se você usar o tipo de dado string.

TIPO DE DADO NORMALIZEDSTRING

O tipo de dado normalizedString é derivado do tipo de dado string.

Ele também contém caracteres, mas o processador XML vai remover quebras de linha, retornos de carro, e caracteres de tabulação.

A seguir está um exemplo de declaração de normalizedString em um esquema:

```
<xs:element name="customer">  
  <xs:complexType>  
    <xs:sequence>  
      <xs:element type="xs:normalizedString"/>  
    </xs:sequence>  
</xs:complexType>  
</xs:element>
```

Um element em seu documento poderia ser assim:

```
<customer>John Smith</customer>
```

Ou assim:

```
<customer>John Smith</customer>
```

Nota: No exemplo acima, o processador XML vai substituir a tabulação por espaços.

TIPO DE DADO TOKEN

O tipo de dado token também é derivado do tipo de dado string.

Ele também contém caracteres, mas o processador XML vai remover quebras de linha, retornos de carro, tabulação, espaços iniciais e finais, e espaços múltiplos.

A seguir está um exemplo de declaração token em um esquema:

```
<xs:element name="customer" type="xs:token"/>
```

Um elemento em seu documento poderia ser assim:

```
<customer>John Smith</customer>
```

Ou assim:

```
<customer> John Smith </customer>
```

Nota: No exemplo acima o processador XML vai remover a tabulação.

TIPOS DE DADOS STRING

Name	Description
ENTITIES	
ENTITY	
ID	Uma string que representa o atributo ID em XML (usado apenas com atributos de schema)
IDREF	Uma string que representa o atributo IDREF em XML (usado apenas com atributos schema)
IDREFS	
language	Uma string que contém a id de uma linguagem válida
Name	Uma string que contém um nome XML válido
NCName	

NMTOKEN	Uma string que representa o atributo NMTOKEN em XML (usado apenas com atributos schema)
NMTOKENS	
normalizedString	Uma string que não contém quebras de linha, retornos de carro, ou tabulação
QName	
string	Uma cadeia de caracteres
token	Um string que não contém quebras de linhas, retornos de carro, tabulação, espaços iniciais e finais, ou múltiplos espaços

RESTRIÇÕES EM TIPOS DE DADOS STRING

Restrições que podem ser usados com tipos de dados string:

- enumeration
- lenght
- maxLength
- minLength
- pattern (NMTOKENS, IDREFS, e ENTITIES não podem usar esta restrição)
- whiteSpace

TIPOS DE DADOS DE DATA E HORA

Tipos de dados de data e hora são usados para valores que contém data e hora.

TIPO DE DADO DATE

O tipo de dado date é usado para especificar uma data.

A data é especificada da seguinte forma "YYYY-MM-DD", onde:

- YYYY indica o ano
- MM indica o mês
- DD indica o dia

Nota: Todos componentes são obrigatórios!

A seguir está um exemplo de declaração uma data em um schema:

```
<xs:element name="start" type="xs:date"/>
```

Um elemento em seu documento poderia ser assim:

```
<start>2002-09-24</start>
```

TIME ZONES

Para especificar um *time zone* (fuso horário), você pode usar uma data no horário UTC adicionando um "Z" depois da data, como aqui:

```
<start>2002-09-24Z</start>
```

ou você pode especificar uma área do horário UTC adicionando um horário positivo ou negativo depois da data, como aqui:

```
<start>2002-09-24-06:00</start> ou <start>2002-09-24+06:00</start>
```

TIPO DE DADO TIME

O tipo de dado time é usado para especificar um horário.

O horário é especificado da seguinte forma "hh:mm:ss", onde:

- hh indica a hora
- mm indica os minutos
- ss indica os segundos

Nota: Todos componentes são obrigatórios!

A seguir está um exemplo de declaração de um time em schema:

```
<xss:element name="start" type="xs:time"/>
```

Um elemento em seu documento poderia ser assim:

```
<start>09:00:00</start>
```

Ou assim:

```
<start>09:30:10.5</start>
```

TIME ZONES

Para especificar um *time zone* (fuso horário), você pode tanto usar um horário em UTC, adicionando um "Z" ao final dele, como aqui:

```
<start>09:30:10Z</start>
```

ou especificando uma área do horário UTC, adicionando um horário positivo ou negativo depois dele, como aqui:

```
<start>09:30:10-06:00</start>  
<start>09:30:10+06:00</start>
```

ou

TIPO DE DADO DATETIME

O tipo de dado *dateTime* é usado para especificar uma data e um horário.

O *dateTime* é especificado da seguinte forma "YYYY-MM-DDThh:mm:ss", onde:

- YYYY indica o ano
- MM indica o mês
- DD indica os dias

- T indica o início da seção obrigatória time
- hh indica a hora
- mm indica os minutos
- ss indica os segundos

Nota: Todos componentes são obrigatórios!

A seguir está um exemplo de uma declaração dateTime em um schema:

```
<xs:element name="stardate" type="xs:dateTime"/>
```

Um elemento em seu documento poderia ser assim:

```
<startdate>2002-05-30T09:00:00</startdate>
```

Ou assim:

```
<startdate>2002-05-30T09:30:10.5</startdate>
```

TIME ZONES

Para especificar um *time zone* (fuso horário), você pode tanto utilizar um dateTime em UTC adicionado de um "Z", como aqui:

```
<startdate>2002-05-30T09:30:10Z</startdate>
```

ou especificando uma área do horário UTC, adicionando um horário positivo ou negativo ao horário, como aqui:

```
<startdate>2002-05-30T09:30:10-06:00</startdate>  
ou <startdate>2002-05-30T09:30:10+06:00</startdate>
```

TIPO DE DADO DURATION

O tipo de dado duration é usado para especificar um intervalo de tempo.

O intervalo de tempo é especificado da seguinte forma "PnYnMnDTnHnMnS", onde:

- P indica o período (obrigatório)
- nY indica o número de anos
- nM indica o número de meses
- nD indica o número de dias
- T indica o início de uma seção de hora (obrigatório se você vai especificar horas, minutos ou segundos)
- nH indica o número de horas
- nM indica o número de minutos
- nS indica o número de segundos

A seguir está um exemplo de uma declaração de duration em um schema:

```
<xs:element name="period" type="duration"/>
```

Um elemento em seu documento poderia ser assim:

```
<period>P5Y</period>
```

O exemplo acima indica um período de cinco anos.

Ou assim:

```
<period>P5Y2M10D</period>
```

O exemplo acima indica um período de cinco anos, dois meses e dez dias.

Ou assim:

```
<period>P5Y2M10DT15H</period>
```

O exemplo acima indica um período de cinco anos, dois meses, dez dias e 15 horas.

Ou assim:

```
<period>PT15H</period>
```

O exemplo acima indica um período de 15 horas.

DURAÇÃO NEGATIVA

Para especificar uma duração negativa, insira um sinal de menos antes de P:

```
<period>-P10D</period>
```

O exemplo acima indica um período de menos 10 dias.

TIPOS DE DADOS DE DATA E HORA

Nome	Descrição
date	Define um valor de data
dateTime	Define um valor de data e hora
duration	Define um intervalo de tempo
gDay	Define uma parte de uma data - o dia (DD)
gMonth	Define uma parte de uma data - o mês (MM)
gMonthDay	Define uma parte de uma data - o mês e o dia (MM-DD)
gYear	Define uma parte de uma data - o ano (YYYY)
gYearMonth	Define uma parte de uma data - o ano e o mês (YYYY-MM)
time	Define um valor de hora

RESTRIÇÕES EM TIPOS DE DADOS DATE

Restrições que podem ser usadas com tipos de dados date:

- enumeration
- maxExclusive
- maxInclusive
- minExclusive
- minInclusive
- pattern
- whiteSpace

TIPOS DE DADOS NUMÉRICOS

Tipos de dados decimais são usados para armazenar números

TIPO DE DADO DECIMAL

O tipo de dado decimal é usado para especificar um valor numérico.

O exemplo a seguir é uma declaração de decimal em um schema:

```
<xs:element name="prize" type="xs:decimal"/>
```

Um elemento em seu documento poderia ser assim:

```
<prize>999.50</prize>
```

Ou assim:

```
<prize>+999.5450</prize>
```

Ou assim:

```
<prize>-999.5230</prize>
```

Ou assim:

```
<prize>0</prize>
```

Ou assim:

```
<prize>14</prize>
```

Nota: O número máximo de dígitos decimais que você pode especificar é 18!

TIPO DE DADO INTEGER

O tipo de dado integer é usado para especificar um valor numérico sem um componente fracionário.

O exemplo a seguir é uma declaração de um integer em um schema:

```
<xss:element name="prize" type="xss:integer"/>
```

Um elemento em seu documento poderia ser assim:

```
<prize>999</prize>
```

Ou assim:

```
<prize>+999</prize>
```

Ou assim:

```
<prize>-999</prize>
```

Ou assim:

```
<prize>0</prize>
```

TIPOS DE DADOS NUMÉRICOS

Note que todos os tipos de dados abaixo derivam do tipo de dado Decimal (exceto ele mesmo)!

Nome	Descrição
byte	Um inteiro 8-bit com sinal
decimal	Um valor decimal
int	Um inteiro 32-bit com sinal
integer	Um valor inteiro
long	Um inteiro 64-bit com sinal
negativeInteger	Um inteiro contendo apenas valores negativos (...,-2,-1)
nonNegativeInteger	Um inteiro contendo apenas valores não negativos (0,1,2,...)
nonPositiveInteger	Um inteiro contendo apenas valores não positivos (...,-2,-1,0)
positiveInteger	Um inteiro contendo apenas valores positivos (1,2,...)
short	Um inteiro 16-bit com sinal
unsignedLong	Um inteiro 64-bit sem sinal
unsignedInt	Um inteiro 32-bit sem sinal
unsignedShort	Um inteiro 16-bit sem sinal
unsignedByte	Um inteiro 8-bit sem sinal

RESTRIÇÕES EM TIPOS DE DADOS NUMÉRICOS

Restrições que podem ser usadas com tipos de dados numéricos:

- enumeration
- fractionDigits
- maxExclusive
- maxInclusive
- minExclusive
- minInclusive
- pattern
- totalDigits
- whiteSpace

TIPOS DE DADOS VARIADOS

Outros tipos de dados variados são boolean, base64Binary, hexBinary, float, double, anyURI, QName, e NOTATION.

TIPO DE DADO BOOLEAN

O tipo de dado boolean é usado para especificar um valor true (verdadeiro) ou false (falso).

O exemplo a seguir é uma declaração de um boolean em um schema:

```
<xs:attribute name="disabled" type="xs:boolean"/>
```

Um elemento em seu documento poderia ser assim:

```
<prize disabled="true">999</prize>
```

Nota: Valores válidos para boolean são true, false, 1 (que indica true), e 0 (que indica false).

TIPOS DE DADOS BINÁRIOS

Tipos de dados binários são usados para expressar dados binários.

Temos dois tipos de dados binários:

- base64Binary (dado binário codificado em Base64)
- hexBinary (dado binário codificado em hexadecimal)

O exemplo a seguir é uma declaração de um hexBinary em um schema:

```
<xs:element name="blobsrc" type="xs:hexBinary"/>
```

TIPO DE DADO ANYURI

O tipo de dado anyURI é usado para especificar uma URI.

O exemplo a seguir é uma declaração de um anyURI em um schema:

```
<xs:attribute name="src" type="xs:anyURI"/>
```

Um elemento em seu documento poderia ser assim:

```
<pic  
src="http://www.soaexpert.com.br/images/smiley.gif" />
```

Nota: Se uma URI tem espaços, substitua-os por %20.

RESTRIÇÕES EM TIPOS DE DADOS VARIADOS

Restrições que podem ser usados com outros tipos de dados:

- enumeration (um tipo de dado Boolean não pode usar esta restrição)
- length (um tipo de dado Boolean não pode usar esta restrição)
- maxLength (um tipo de dado Boolean não pode usar esta restrição)
- minLength (um tipo de dado Boolean não pode usar esta restrição)
- pattern
- whiteSpace

13.0 XPath Conceitos

XPath é um conjunto de regras de sintaxe para manipular partes de um documento XML.

XPath é o elemento principal no padrão W3C XSLT. Sem o conhecimento de XPath você não será capaz de criar documentos XSLT - XQuery.

Para entender um pouco mais sobre **XPath**, é necessário conhecer alguns conceitos correlacionados.

Nós: Todo documento XML é formado por nós. Temos o nó raiz, nós intermediários, nós dentro de outros nó etc.

Tipos de Nós: É intuitivo pensar que exista apenas um tipo de nó: Os elementos. Mas para a especificação XML, os elementos não são os únicos tipos de nós existentes. Podemos nos deparar com nós do tipo elementos, atributos, comentários, instruções de processamento, etc.

No exemplo abaixo temos um tipo de nó elemento (Site) e um tipo de nó atributo (URL).

Ex:

```
<Site URL="http://www.xml.org"> Applying XML and Web Services Standards </Site>.
```

Neste caso o valor do nó "Site" é "**Applying XML and Web Services Standards**" e do nó "URL" é "<http://www.xml.org>". Quando utilizamos atributos na XPath, devemos especificar o @ antes do nome do atributo.

Localização: A navegação em um documento XML é muito parecida com a navegação em uma estrutura de diretórios ou nos hyperlinks que usamos na internet. Quando desejamos acessar um determinado arquivo, a localização desse arquivo é composta por todas as subpastas que tivemos que navegar até chegar a esse arquivo. Ex: C:\Projetos\XML\XPath\Livros.xml

Se fosse uma página de Internet talvez tivessemos algo do tipo <http://localhost/Projetos/XML/XPath/Livros.XML>. Com a XPath a localização seria expressa na forma **/Projetos/XML/XPath/** e o arquivo XML seria:

```
<Projetos>
<XML>
  <XPATH>Livros.xml</XPATH>
</XML>
</Projetos>
```

Eixos: Um documento XML é uma representação de uma **árvore**. Nessa árvore existem nós pais, nós filhos, nós irmãos, etc. No documento **XML** abaixo podemos perceber algumas dessas relações. Veja que "Materiais" é o nó raiz, "Material" é filho de "Materiais" e ao mesmo tempo pai de "Nome", "Valor" e "Cores".

Perceba também que "Nome", "Valor" e "Cores" são nós irmãos. Os eixos são essas referência e representam coordenadas para o tipo de nó a ser utilizado na XPath.

```
<Materiais>
<Material>
  <Nome>Caneta</Nome>
  <Valor>2</Valor>
  <Cores>
    <Cor>Azul</Cor>
    <Cor>Verde</Cor>
  </Cores>
<Material>
</Materiais>
```

Expressões: Os comandos utilizados na XPath são conhecidos como expressões XPath. Essas expressões são compostas de informações de localização dos nó a ser pesquisado, funções utilizadas etc. Em linhas gerais ela é a instrução que fará a pesquisa (como se fosse uma instrução SQL). Ex: **/Materiais/Material/Nome**

Funções e variáveis: A XPath dispõe de diversas funções básicas que são encontradas na maioria das linguagens de programação. Ex: substring, round, etc.

ResultSet: O resultado de uma consulta XPath, em princípio, deve ser uma coleção de nós que também representa um documento XML.

13.2 Localizando Nós

Documentos XML podem ser representados como árvores de nós (muito similar à visão em árvore das pastas do seu computador).

XPath usa uma expressão padrão para identificar nós em um documento XML. Um padrão XPath é uma lista nomes de elementos filhos separada por barras que descreve o caminho através do documento XML. O padrão "seleciona" elementos que combinam com o caminho.

A expressão XPath a seguir seleciona todos os elementos preço de todos os elementos cd do elemento catalog:

```
/catalog/cd/price
```

Nota: Se o caminho começa com uma barra (/) ele representa um caminho absoluto para um elemento!

Nota: Se o caminho começa com duas barras (//) então todos os elementos no documento que se encaixam no critério serão selecionados (mesmo que eles estejam em níveis diferentes da árvore XML)!

A expressão XPath a seguir seleciona todos os elementos cd no documento:

```
//cd
```

SELECIONANDO ELEMENTOS DESCONHECIDOS

Curingas (*) podem ser usados para selecionar elementos XML desconhecidos.

A expressão XPath a seguir seleciona todos os elementos filhos de todos os elementos cd do elemento catalog:

```
/catalog/cd/*
```

A expressão XPath a seguir seleciona todos os elementos price que são elementos netos do elemento catalog:

```
/catalog/*/price
```

A expressão XPath a seguir seleciona todos os elementos price que têm dois ancestrais:

```
/*/*/price
```

A expressão XPath a seguir seleciona todos os elementos no documento:

```
//*[1]
```

SELECIONANDO SEÇÕES

Usando-se colchetes numa expressão XPath você pode especificar um elemento adiante.

A expressão XPath a seguir seleciona o primeiro elemento cd filho do elemento catalog:

```
/catalog/cd[1]
```

A expressão XPath a seguir seleciona o último elemento cd filho do elemento catalog (*Nota: não existe a função first()*):

```
/catalog/cd[last()]
```

A expressão XPath a seguir seleciona todos os elementos cd do elemento catalog que tem um elemento price:

```
/catalog/cd[price]
```

A expressão XPath a seguir seleciona todos os elementos cd do elemento catalog que tem um elemento price com valor de 10.90:

```
/catalog/cd[price=10.90]
```

A expressão XPath a seguir seleciona todos os elementos price de todos os elementos cd do elemento catalog que tem um elemento price com valor de 10.90:

```
/catalog/cd[price=10.90]/price
```

SELECIONANDO VÁRIOS CAMINHOS

Usando o operador "|" numa expressão XPath você pode selecionar vários caminhos.

A expressão XPath a seguir seleciona todos os elementos title e artist do elemento cd do elemento catalog:

```
/catalog/cd/title | /catalog/cd/artist
```

A expressão XPath a seguir seleciona todos os elementos title e artist do documento:

```
//title | //artist
```

A expressão XPath a seguir seleciona todos os elementos title, artist e price do documento:

```
//title | //artist | //price
```

A expressão XPath a seguir seleciona todos os elementos title do elemento cd do elemento catalog, e todos os elementos artist no documento:

```
/catalog/cd/title | //artist
```

SELECIONANDO ATRIBUTOS

Em XPath todos os atributos são especificados pelo prefixo "@". Esta expressão XPath seleciona todos os atributos chamados country:

```
//@country
```

Esta expressão XPath seleciona todos os elementos cd que tem um atributo chamado country:

```
//cd[@country]
```

Esta expressão XPath seleciona todos os elementos cd que tem algum atributo:

```
//cd[@*]
```

Esta expressão XPath seleciona todos os elementos cd que tem um atributo chamado country com valor 'UK':

```
//cd[@country='UK']
```

13.3 Location PATHS – Caminhos Locais

Uma expressão de caminho local (location path) resulta em um conjunto de nós. Um caminho local pode ser absoluto ou relativo.

Um caminho local absoluto começa com uma barra (/) e um caminho local relativo não. Em ambos os casos o caminho local consiste de um ou mais níveis de localização, cada um separado por uma barra:

Um caminho local absoluto:	/step/step/...
Um caminho local relativo:	step/step

Os níveis de localização são avaliados em ordem um de cada vez, da esquerda pra direita. Cada nível é avaliado segundo os nós no conjunto de nós atual. Se o caminho local é absoluto, o conjunto de nós atual é o nó raiz. Se o caminho local é relativo, o conjunto de nós atual consiste do nó onde a expressão está sendo usada. Níveis de localização consistem de:

- um eixo (especifica a relação de árvore entre os nós selecionados pelo nível de localização e o nó atual)
- um nó teste (especifica o tipo de nó e o nome expandido dos nós selecionados pelo nível de localização)
- zero ou mais predicados (usa expressões para refinar mais o conjunto de nós selecionado pelo nível de localização)

A sintaxe para um nível de localização é:

nomeeixo::noteste[predicado]

Exemplo:

child::price[price=9.90]

EIXOS E NÓS DE TESTE

Um eixo define um conjunto de nós relativo ao nó atual. Um nó de teste é usado para identificar um nó dentro de um eixo. Nós podemos executar um nó de teste por nome ou por tipo.

Nome do Eixo	Descrição
ancestor	Contém todos os ancestrais (pais, avós, etc) do nó atual. Nota: Este eixo incluirá sempre o nó raiz, a menos que o nó atual seja o nó raiz
ancestor-or-self	Contém o nó atual mais todos os seus ancestrais (pai, avô, etc)

attribute	Contém todos os atributos do nó atual
child	Contém todos os filhos do nó atual
descendant	Contém todos os descendentes (filhos, netos, etc) do nó atual. Nota: Este eixo nunca contém atributos ou nós namespace
descendant-or-self	Contém o nó atual mais todos os seus descendentes (filhos, netos, etc)
following	Contém tudo no documento depois da tag de fechamento do nó atual
following-sibling	Contém todos os irmãos depois do nó atual. Nota: Se o nó atual é um nó atributo ou um nó namespace, este eixo estará vazio
namespace	Contém todos os nós namespace do nó atual
parent	Contém o pai do nó atual
preceding	Contém tudo no documento que está antes da tag de abertura do nó atual
preceding-sibling	Contém todos os irmãos antes do nó atual. Nota: Se o nó atual é um nó atributo ou um nó namespace, este eixo estará vazio
self	Contém o nó atual

Exemplos

Exemplo	Resultado
child::cd	Seleciona todos os elementos cd que são filhos do nó atual (se o nó atual não tem cds filhos, será selecionado um conjunto vazio de nós)
attribute::src	Seleciona o atributo src do nó atual (se o nó atual não tem atributo src, será selecionado um conjunto vazio de nós)
child::*	Seleciona todos os elementos filhos do nó atual
attribute::*	Seleciona todos os atributos do nó atual
child::text()	Seleciona o nó texto filho do nó atual
child::node()	Seleciona todos os filhos do nó atual
descendant::cd	Seleciona todos os elementos cd descendentes do nó atual
ancestor::cd	Seleciona todos os cds ancestrais do nó atual
ancestor-or-self::cd	Seleciona todos os cds ancestrais do nó atual e, se o nó atual é um elemento cd, seleciona o nó atual também
child::* / child::price	Seleciona todos os preços netos do nó atual
/	Seleciona a raiz do documento

PREDICADOS

Um predicado filtra um conjunto de nós em um novo conjunto de nós. Um predicado fica dentro de colchetes ([]).

Exemplos

Exemplo	Resultado
child::price[price=9.90]	Seleciona todos os elementos price que são filhos do nó atual com um preço igual a 9.90
child::cd[position()=1]	Seleciona o primeiro cd filho do nó atual
child::cd[position()=last()]	Seleciona o último cd filho do nó atual
child::cd[position()=last()-1]	Seleciona o penúltimo cd filho do nó atual
child::cd[position()<6]	Seleciona os primeiros cinco cds filhos do nó atual
/descendant::cd[position()=7]	Seleciona o sétimo elemento cd no documento
child::cd[attribute::type="classic"]	Seleciona todos os cds filhos do nó atual que têm um atributo tipo com o valor "classic"

SINTAXE ABREVIADA DOS CAMINHOS LOCAIS

Abreviações podem ser usadas para descrever um caminho local.

A abreviação mais importante é que child:: pode ser omitido de um nível de localização.

Abrev.	Significado	Exemplo
nada	child::	cd é o mesmo que child::cd
@	attribute::	cd[@type="classic"] é o mesmo que child::cd[attribute::type="classic"]
.	self::node()	.//cd é o mesmo que self::node() / descendant-or-self::node() / child::cd
..	parent::node()	../cd é o mesmo que parent::node() / child::cd
//	/descendant-or-self::node()	cd é o mesmo que /descendant-or-self::node() / child::cd

Exemplos

Exemplo	Resultado
cd	Seleciona todos os elementos cd que são filhos do nó atual
*	Seleciona todos os elementos filhos do nó atual
text()	Seleciona todos os nós textos filhos do nó atual
@src	Seleciona o atributo src do nó atual
@*	Seleciona todos os atributos do nó atual
cd[1]	Seleciona o primeiro cd filho do nó atual
cd[last()]	Seleciona o último cd filho do nó atual

*/cd	Seleciona todos os cds netos do nó atual
/book/chapter[3]/para[1]	Seleciona o primeiro parágrafo do terceiro capítulo do livro
//cd	Seleciona todos os elementos cds descendentes da raiz do documento e assim seleciona todos os elementos cds no mesmo documento como o nó atual
.	Seleciona o nó atual
.//cd	Seleciona os elementos cds descendentes do nó atual
..	Seleciona o pai do nó atual
../@src	Seleciona o atributo src do pai do nó atual
cd[@type="classic"]	Seleciona todos os cds filhos do nó atual que têm o atributo "type" com o valor "classic"
cd[@type="classic"][5]	Seleciona o quinto cd filho do nó atual que tem o atributo "type" com o valor "classic"
cd[5][@type="classic"]	Seleciona o quinto cd filho do nó atual se esse filho tem o atributo "type" com valor "classic"
cd[@type and @country]	Seleciona todos os cds filhos do nó atual que têm ambos os atributos "type" e "country"

13.4 Expressões XPATH

XPath suporta expressões numéricas, de igualdade, relacionais e booleanas.

EXPRESSÕES NUMÉRICAS

Expressões numéricas são usadas para realizar operações aritméticas em números.

Operador	Descrição	Exemplo	Resultado
+	Adição	6 + 4	10
-	Subtração	6 - 4	2
*	Multiplicação	6 * 4	24
div	Divisão	8 div 4	2
mod	Módulo (resto da divisão)	5 mod 2	1

Nota: XPath sempre converte cada operando em um número antes de realizar um expressão aritmética.

EXPRESSÕES DE IGUALDADE

Expressões de igualdade são usadas para testar a igualdade entre dois valores.

Operador	Descrição	Exemplo	Resultado
=	Igual	price=9.80	true (se o preço é igual a 9.80)
!=	Diferente	price!=9.80	false

TESTANDO UM CONJUNTO DE NÓS

Se é testada a igualdade do valor de teste em relação a um conjunto de nós, o resultado é verdadeiro se o conjunto de nós contém algum nó com um valor igual ao valor de teste.

Se é testada a desigualdade do valor de teste em relação a um conjunto de nós, o resultado é verdadeiro se o conjunto de nós contém algum nó com um valor diferente ao valor de teste.

O resultado é que o conjunto de nós pode ser igual e diferente ao mesmo tempo!!!

EXPRESSÕES RELACIONAIS

Expressões relacionais são usadas para comparar dois valores.

Operador	Descrição	Exemplo	Resultado
<	Menor	price<9.80	false (se o preço é 9.80)
<=	Menor ou igual	price<=9.80	true
>	Maior	price>9.80	false
>=	Maior ou igual	price>=9.80	true

Nota: XPath sempre converte cada operando em um número antes de fazer a avaliação.

EXPRESSÕES BOOLEANAS

Expressões booleanas são usadas para comparar dois valores.

Operador	Descrição	Exemplo	Resultado
or	Ou	price=9.80 or price=9.70	true (se o preço é 9.80)
and	E	price<=9.80 and price=9.70	false

13.5 Funções XPATH

XPath contém uma biblioteca de funções para conversão de dados.

BIBLIOTECA DE FUNÇÕES XPATH

A biblioteca de funções XPath contém um conjunto de funções centrais para conversão e tradução de dados.

FUNÇÕES PARA CONJUNTO DE NÓS

Nome	Descrição	Sintaxe
count()	Retorna o número de nós num conjunto de nós	number=count(node-set)
id()	Seleciona elementos pelo seu ID único	node-set=id(value)
last()	Retorna o número da posição do último nó na lista de nós processados	number=last()
local-name()	Retorna a parte local de um nó. Um nó normalmente consiste de um prefixo, os dois pontos, seguidos pelo nome	string=local-name(node)

	local	
name()	Retorna o nome do nó	string=name(node)
namespace-uri()	Retorna o namespace URI de um nó específico	uri=namespace-url(node)
position()	Retorna a posição na lista de nós do nó que está sendo processado atualmente	number=position()

FUNÇÕES NUMÉRICAS

Nome	Descrição	Sintaxe	Exemplo
ceiling()	Retorna o menor inteiro que não é menor que o argumento (função teto)	number=ceiling(number)	ceiling=(3.14) -> Resultado: 4
floor()	Retorna o maior inteiro que não é maior que o argumento (função chão)	number=floor(number)	floor(3.14) -> Resultado: 3
number()	Converte o argumento em um	number=number(value)	number('100') -> Resultado: 100

	número		
round()	Arredonda o argumento para o inteiro mais próximo	integer=round(number)	round(3.14) -> Resultado: 3
sum()	Retorna o valor total de um conjunto de valores numéricos num conjunto de nós	number=sum(nodeset)	sum(/cd/price)

FUNÇÕES BOOLEANAS

Nome	Descrição	Sintaxe
boolean()	Converte o argumento 'value' em Booleano e retorna verdadeiro ou falso	bool=boolean(value)
false()	Retorna falso	false() Exemplo: number(false()) -> Resultado: 0
lang()	Retorna verdadeiro se o argumento 'language' é	bool=lang(language)

	igual ao idioma do elemento xsl:lang, senão retorna falso	
not()	Retorna verdadeiro se o argumento 'condition' é falso, e falso se o argumento é verdadeiro	bool=not(condition) Exemplo: not(false())
true()	Retorna verdadeiro	true() Exemplo: number(true()) -> Resultado: 1

13.6 Exercícios XPATH

Nós iremos usar o catálogo de CDs XML:

XML: (Uma fração do catalogo)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<catalog>  <cd>  <title>Empire Burlesque</title>
<artist>Bob Dylan</artist>  <country>USA</country>
<company>Columbia</company>
<price>10.90</price>  <year>1985</year>  </cd>
<cd>  <title>Hide your heart</title>
<artist>Bonnie Tyler</artist>
<country>UK</country>  <company>CBS
Records</company>  <price>9.90</price>
<year>1988</year>  </cd> . . . </catalog>
```

Utilizar o arquivo: cdcatalog.xml

SELECIONANDO Nós

Selecione nós de um documento XML usando a função selectNodes
Esta função tem uma expressão de caminho local como argumento:

Xpath expression

SELECIONANDO Nós CD

Selecione somente o primeiro nó do catálogo de cds:

/catalog/cd[0]

SELECIONANDO Nós PRICE

Selecione todos os nós price do catálogo de cds:

catalog/cd/price

SELECIONANDO O TEXTO DOS Nós PRICE

Selecione apenas o texto dos nós price:

/catalog/cd/price/text()

SELECIONANDO Nós CD COM PRICE>10.80

Selecione todos os nós cd com price >10.80:

/catalog/cd[price>10.80]

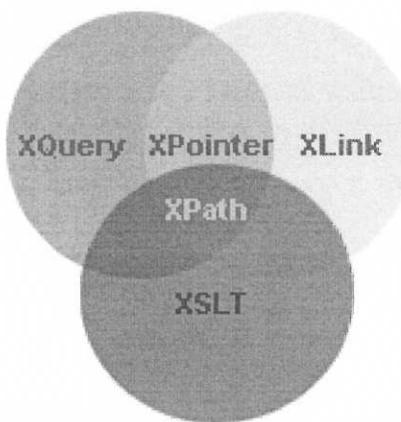
SELECIONANDO Nós PRICE COM PRICE>10.80

Selecione todos os nós price com price > 10.80:

/catalog/cd[price>10.80]/price

14.0 XQuery FastTrack

O que é XQuery ?



XQuery é uma linguagem para buscar e extrair elementos e atributos de documentos XML.

XQuery é construída em cima de expressões XPATH.

XQuery é suportada por um grande número de banco de dados.

XQuery and XPath

XQuery 1.0 e XPath 2.0, compartilham o mesmo conjunto de dados e suporte às mesmas funções e operadores.

Nós usaremos o seguinte original de XML no exemplo abaixo.
“bookdetails.xml”:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bookstore>
<book category="COOKING">
<title lang="en">Everyday Italian</title>
<author>Giada De Laurentiis</author>
<year>2005</year>
<price>30.00</price>
</book>
<book category="CHILDREN">
<title lang="en">Harry Potter</title>
<author>J K. Rowling</author>
<year>2005</year>
<price>29.99</price>
</book>
<book category="WEB">
<title lang="en">XQuery Kick Start</title>
<author>James McGovern</author>
<author>Per Bothner</author>
<author>Kurt Cagle</author>
<author>James Linn</author>
<author>Vaidyanathan Nagarajan</author>
<year>2003</year>
<price>49.99</price>
</book>
<book category="WEB">
<title lang="en">Learning XML</title>
<author>Erik T. Ray</author>
<year>2003</year>
<price>39.95</price>
</book>
</bookstore>
```

14.1 Como Selecionar Nós

Funções

XQuery usa funções, para extrair os dados do XML. A função **doc()** é usada basicamente o arquivo “bookdetails.xml”:

```
doc("bookdetails.xml")
```

Expressões PATH

XQuery usa expressões PATH através dos elementos XML.

A seguinte expressão é usada para selecionar todos os elementos **título** contidos em “bookdetails.xml”:

```
doc("books.xml")/bookstore/book/title
```

Resultado :

```
<title lang="en">Everyday Italian</title>
<title lang="en">Harry Potter</title>
<title lang="en">XQuery Kick Start</title>
<title lang="en">Learning XML</title>
```

Predicados

XQuery pode usar predicados para limitar os dados que serão extraídos à uma regra específica, neste caso o “elemento” preço, deve possuir o **valor** menor que 30.

```
doc("books.xml")/bookstore/book[price<30]
```

Resultado:

```
<book category="CHILDREN">
  <title lang="en">Harry Potter</title>
  <author>J K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>
```

14.2 Expressões FLWOR (“Flower”)

FLWOR é um acrônimo para “For, Let, Where, Order by , Return”, também conhecido como Flower.

Selecionando nós com FLOWR:

```
for $x in doc("books.xml")/bookstore/book
where $x/price>30
return $x/title
```

Resultado:

```
<title lang="en">XQuery Kick Start</title>
<title lang="en">Learning XML</title>
```

Com FLWOR você pode ordenar o resultado:

```
for $x in doc("books.xml")/bookstore/book
where $x/price>30
order by $x/title
return $x/title
```

A cláusula **for** , seleciona todos os elementos **book** e armazena numa variável chamada **\$x**.

O **where** seleciona somente elementos **book** com o valor do elemento **price** **maior** que 30.

O **order by** define a ordem que os elementos devem aparecer e o **return** especifica quais eleementos devem ser retornados.

Resultado:

```
<title lang="en">Learning XML</title>
<title lang="en">XQuery Kick Start</title>
```

14.3 XQuery Sintaxe

Algumas regras básicas da linguagem:

- XQuery é **case-sensitive**
- XQuery elementos, atributos e variáveis dever ter nomes XML
- valor de uma string XQuery pode conter aspas **simples** ou **duplas**.
- Uma variável XQuery é definida com \$ seguida pelo nome, exemplo: **\$bookstore**.
- Comentários XQuery são delimitados pelo uso de (: e :). Exemplo:
(: Isso é um comentário :).

14.4 Expressões Condicionais

Expressões: “If-Then-Else” são suportadas em XQuery.

```
for $x in doc("books.xml")/bookstore/book
return if ($x/@category="CHILDREN")
then <child>{data($x/title)}</child>
else <adult>{data($x/title)}</adult>
```

Parentesis ao redor da expressão IF são requeridos. Else é requerido, mas poderia ser somente **else()**.

Resultado:

```
<adult>Everyday Italian</adult>
<child>Harry Potter</child>
<adult>Learning XML</adult>
<adult>XQuery Kick Start</adult>
```

14.5 Comparações

Em XQuery temos duas maneiras de comparar valores:

1 - Comparações gerais: =, !=, <, <=, >, >=

2- Comparações de valor: eq, ne, lt, le, gt, ge

A diferença entre ambos:

A expressão seguinte retorna TRUE se qualquer atributo q possui valor maior que 10:

```
$bookstore//book/@q > 10
```

A segunda expressão, retorna TRUE somente se um atributo q retornado pela expressão for maior que 10. Se existirem mais do que um, um erro acontece.

```
$bookstore//book/@q gt 10
```

14.6 Adicionando Elementos e Atributos

Adição de Elementos HTML e Texto:

```
<html>
<body>

<h1>Bookstore</h1>

<ul>
{
for $x in doc("books.xml")/bookstore/book
order by $x/title
return <li>{data($x/title)} . Category:
{data($x/@category)}</li>
}
</ul>

</body>
</html>
```

Resultado:

```
<html>
<body>

<h1>Bookstore</h1>

<ul>
<li>Everyday Italian. Category: COOKING</li>
<li>Harry Potter. Category: CHILDREN</li>
<li>Learning XML. Category: WEB</li>
<li>XQuery Kick Start. Category: WEB</li>
</ul>

</body>
</html>
```

Neste exemplo, utilizaremos o atributo category como um atributo class da lista HTML:

```
<html>
<body>

<h1>Bookstore</h1>

<ul>
{
for $x in doc("books.xml")/bookstore/book
order by $x/title
return <li
class="{data($x/@category)}">{data($x/title)}</li>
}
</ul>

</body>
</html>
```

Resultado:

```
<html>
<body>
<h1>Bookstore</h1>

<ul>
<li class="COOKING">Everyday Italian</li>
<li class="CHILDREN">Harry Potter</li>
<li class="WEB">Learning XML</li>
<li class="WEB">XQuery Kick Start</li>
</ul>

</body>
</html>
```

14.7 Selecionando e Filtrando Elementos

Analise a seguinte expressão FLWOR:

```
for $x in doc("books.xml")/bookstore/book
where $x/price>30
order by $x/title
return $x/title
```

- For – relaciona a variável à cada item retornado na expressão XPath.
- Where – especifica um critério
- Order by – especifica uma ordenação do resultado
- Return – especifica o que deverá ser o resultado.

FOR:

A cláusula for liga a variável à cada item retornado pela expressão, resultando numa iteração. Você pode ter múltiplas cláusulas for numa mesma expressão, assim como nas outras linguagens de programação.

```
for $x in (1 to 5)
return <test>{$x}</test>
```

Resultado:

```
<test>1</test>
<test>2</test>
<test>3</test>
<test>4</test>
<test>5</test>
```

A palavra-chave **at** é usada como contador para a iteração:

```
for $x at $i in doc("books.xml")/bookstore/book/title
return <book>{$i}. {data($x)}</book>
```

Resultado:

```
<book>1. Everyday Italian</book>
<book>2. Harry Potter</book>
<book>3. XQuery Kick Start</book>
<book>4. Learning XML</book>
```

Você pode ter mais de uma expressão num loop for, separados por vírgulas:

```
for $x in (10,20), $y in (100,200)
return <test>x={$x} and y={$y}</test>
```

Resultado:

```
<test>x=10 and y=100</test>
<test>x=10 and y=200</test>
<test>x=20 and y=100</test>
<test>x=20 and y=200</test>
```

LET:

A cláusula **let** permite atribuir variáveis evitando ciclos de repetição da mesma expressão.

```
let $x := (1 to 5)
return <test>{$x}</test>
```

Resultado:

```
<test>1 2 3 4 5</test>
```

ORDER:

A cláusula order é usada para especificar a ordem de um resultado. No exemplo abaixo, queremos que o resultado seja ordenado por category e title:

```
for $x in doc("books.xml")/bookstore/book
order by $x/@category, $x/title
return $x/title
```

Resultado:

```
<title lang="en">Harry Potter</title>
<title lang="en">Everyday Italian</title>
<title lang="en">Learning XML</title>
<title lang="en">XQuery Kick Start</title>
```

RETURN:

A cláusula retorno especifica o que deverá ser retornado.

```
for $x in doc("books.xml")/bookstore/book
return $x/title
```

14.8 XQuery Functions

A **XQuery** possui mais de 100 funções incorporadas, como String, Numéricas, comparação de datas etc.

Para conhecer todas as funções:

<http://www.w3.org/TR/xpath-functions/>

O prefixo default para as funções é fn: (namespace).

XQuery – Funções Customizadas:

Se você não conseguiu achar uma função para sua necessidade, você também tem a opção de escrever uma.

As funções customizadas (User-defined functions) pode ser definidas numa query ou numa biblioteca externa.

Sintaxe:

```
declare function prefix:function_name($parameter AS datatype)
AS returnDatatype
{
...codigo aqui da função !
}
```

Regras:

- Utilize a keyword “declare function”
- O nome da function deve ser prefixado
- O tipo dos dados de parâmetros, são os mesmos definidos no XML Schema
- O corpo da função deve estar entre os colchetes.

Exemplo de uma função customizada:

```
declare function local:minPrice($p as xs:decimal?,$d as xs:decimal?)
AS xs:decimal?
{
let $disc := ($p * $d) div 100
return ($p - $disc)
}
```

A utilização da função:

```
<minPrice>{local:minPrice($book/price,$book/discount)}</minPrice>
```

15.0 SOA Security

Em uma estratégia SOA, segurança é um dos principais problemas a serem resolvidos e deve ser suportado por uma fundação infraestrutura sólida. O OASIS está trabalhando num certo número de especificações e perfis para promover a segurança, interoperabilidade entre vendedores de segurança.

Segurança de web services é um conjunto de especificações da OASIS que endereça aspectos ligados à SOAP e XML *binding* de segurança da informação . As maiores questões a serem resolvidas são:

- autenticação, autorização e confidencialidade.

Um efeito colateral direto à problemática é o *Single Sign On*, pois SOA tal como EAI, toca muitos sistemas e esbarra com a segurança de cada um deles (*security silos*).

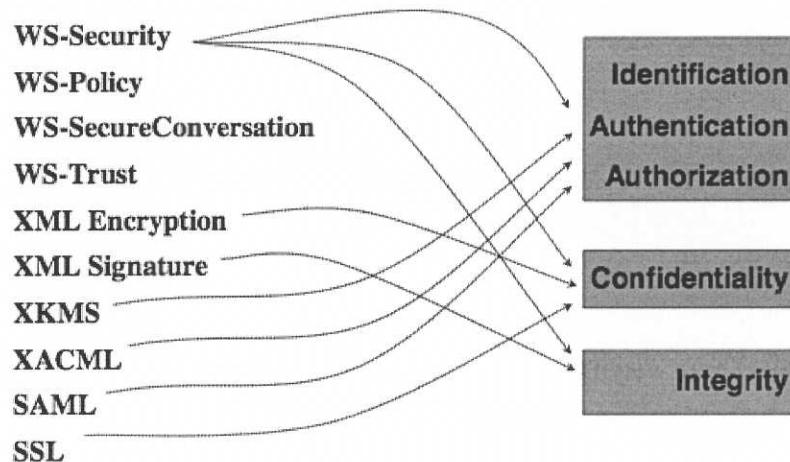
Authentication

Autenticação significa que o serviço que chamou deve prover credenciais para provar sua identidade junto ao serviço destino. Por ora, o suporte à autenticação do SOAP é limitada, por exemplo, SOAP versão 1.2 não inclui um padrão para passar credenciais. Contudo, existem soluções como Security Assertion Markup Language (SAML) desenvolvido pela OASIS que provisiona um framework para troca de autorização e autenticação de informações, adicionando assinatura nas mensagens num documento SOAP.

Authorization

Autorização é o mecanismo usado para consentir acesso a um recurso específico. Novamente, frameworks como SAML ou Java Authentication and Authorization – JASS, podem ser usado como “best-of-breed” a fim de oferecer segurança de infraestrutura para o SOA.

15.1 Security Specification and Standards



WS-Security descreve como anexar assinatura e criptografia no cabeçalho das mensagens SOAP. Em adição, descreve como anexar *tokens* de segurança, incluindo token binário como o certificado X.509 e tickets Kerberos para mensagens.

WS-Policy representa um conjunto de especificações que descrevem as capacidades e restrições das políticas de segurança em intermediários e *endpoints*, por exemplo, token de segurança, suporte à criptografia, regras de privacidade; e como associar essas políticas aos serviços e *endoints*.

WS-SecureConversation descreve como gerenciar e autenticar troca de mensagens entre parceiros, incluindo troca de contexto e estabelecimento e derivação de sessões.

WS-Trust descreve um framework para cofiança que possibilita WebServices seguramente interoperar entre os requests.

15.2 Transport-Level Security

Atualmente é possível assegurar SOAP na comunicação sob HTTP usando o protocolo SSL – *Secure Sockets Layer*. SSL disponibiliza criptografia e assinatura digital tanto para mensagens SOAP quanto para a forma padrão de requests e responses. Entretanto, SSL possui duas grandes limitações que o torna inviável para assegurar comunicação WebService numa aplicação orientada a serviços.

Primeiramente, SSL é desenhado pra comunicação point-to-point. Contudo, serviços orientado à WebServices fazem troca de mensagens SOAP entre dois ou mais *endpoints* (Coreografia ou Orquestração) e requerem uma solução segura para suportar múltiplos *endpoints*.

Segundo, o protocolo SSL está fortemente ligado ao protocolo HTTP. A tecnologia de webservices é especificada de forma neutra à protocolos e suporta diversos, incluindo TCP e SMTP. Portanto, o SSL simplesmente limitaria muito a tecnologia dos webservices.

15.3 Security Specification and Standards

Segurança em nível de mensagem facilita a proteção da mensagem durante a troca de dados entre as aplicações.

Até 2004 não havia nenhuma tecnologia a fim de resolver o problema, então os desenvolvedores optavam em utilizar proteção via SSL-TLS em cima de HTTP ou desenvolviam sua solução customizada, sacrificando a interoperabilidade do processo.

Em abril de 2004 a especificação WS-Security foi estabelecida e aprovada pela OASIS como um padrão aberto.

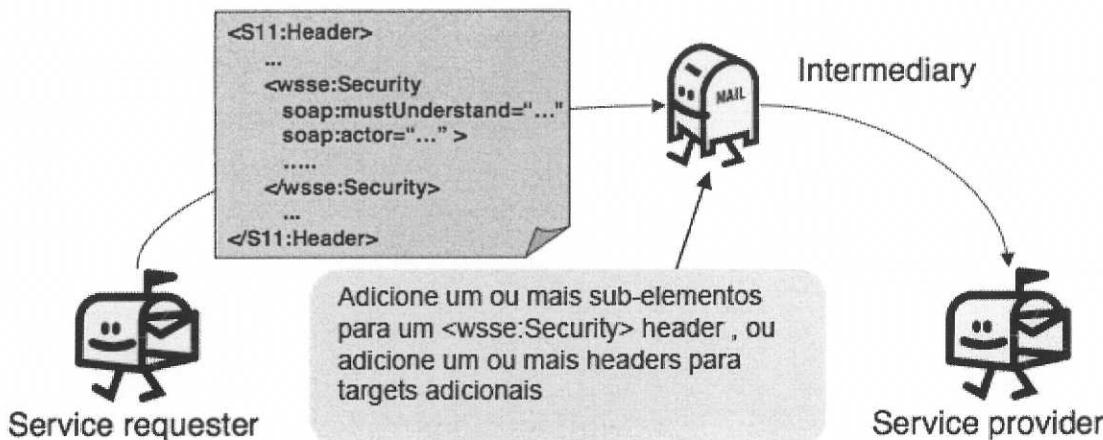
15.4 WS-Security

WS-Security é uma especificação que pode ser utilizada por outras extensões WebServices e protocolos para acomodar uma variedade de modelos de segurança e tecnologias de criptografia.

Suporta uma variedade de modelos, múltiplos formatos de token, de domínios confiáveis, múltiplos tipos de assinatura digital e criptografia. Também é extensível para novos tipos de segurança que vão entrando no mercado como Kerberos ticket ou X.509.

Header Block

WS-Security define através do Header SOAP , o mecanismo para anexar a segurança relacionada à informação que será trafegada.



Ao invés de separar o contexto entre os participantes como SSL-TLS, o WS-Security permite que o contexto seja compartilhado entre a iteração, funcionando como espécie de “guarda-chuva”.

Uma mensagem pode ter múltiplos blocos Header de segurança **<wsse:Security:>** se eles estão destinados à recepientes separados. Dois header blocks não deve possuir o mesmo valor.

Quando um header **<wsse:Security>** inclui um atributo **mustUnderstand=True**, o receptor deve gerar um SOAP fault se não implementa o WSS:SOAP Message Security correspondente.

“Implementação significa a habilidade para interpretar o schema seguindo as regras de processamento especificadas no WSS:SOAP Message Security.”

O receiver deve gerar uma fault, se não estiver apto a interpretar ou processar tokens de segurança contendo bloco header **<wsse:Security>** de acordo com os token profiles.

WS-Security e SOAP

WS-Security provisiona 4 reforços para a mensagem SOAP:

- 1- Um mecanismo de propósito geral para associação de tokens de segurança às mensagens.
- 2- Confidencialidade por incorporação de criptografia do XML.
- 3- Integridade da mensagem, garantindo que essa não sofrerá modificações, através de Assinatura digital com tokens de segurança.
- 4- Um mecanismo para encoding de tokens binários, como certificados **X.509** e tickets **Kerberos**, bem como a inclusão de **chaves criptografadas**.

A especificação *WS-Security* define ainda o conjunto dos seguintes tokens:

- Unsigned security token: **<wsse:UserNameToken>**
 - o Username: **<wsse:Username>**
 - o Password: **<wsse:Password>**
 - o **<wsse:PasswordDigest>**
- Signed security tokens: **<wsse:BinarySecurityToken>**
 - o X.509 - binário
 - o Kerberos ticket - binário

Os dois tipos de passwords:

- PasswordText – Indica que o password é um texto.
- PasswordDigest – Indica que é um encoding Base64, valor SHA-1 hash UTF-8 criografado como password.

```
<wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext">
  <wsse:UsernameToken>
    <wsse:Username>colette</wsse:Username>
    <wsse:Password Type="wsse:PasswordDigest">
      XYZabc123
    </wsse:Password>
    <wsse:Nonce>h52sI9pKV0BVRPUo1QC7Cg=</wsse:Nonce>
  </wsse:UsernameToken>
  ...
</wsse:Security>
```

Há ainda dois elementos opcionais ainda : **wsse:Nonce** e **wsu:Created**.

Um **Nonce** é um valor randômica criado enquanto o **Created** é baseado no timestamp. Se ambos tiverem presentes, o algorítmico para computar o **PasswordDigest** fica assim:

Password_Digest = Base64(SHA-1(nonce+created+password))

PasswordDigest é a melhor maneira para enviar um username e password através de canais não seguros, se não há outros mecanismos aplicados.

PasswordText deve ser usado quando o elemento é criptografado usando **xml-encryption**.

Tokens Binários

O token binário tem dois atributos:

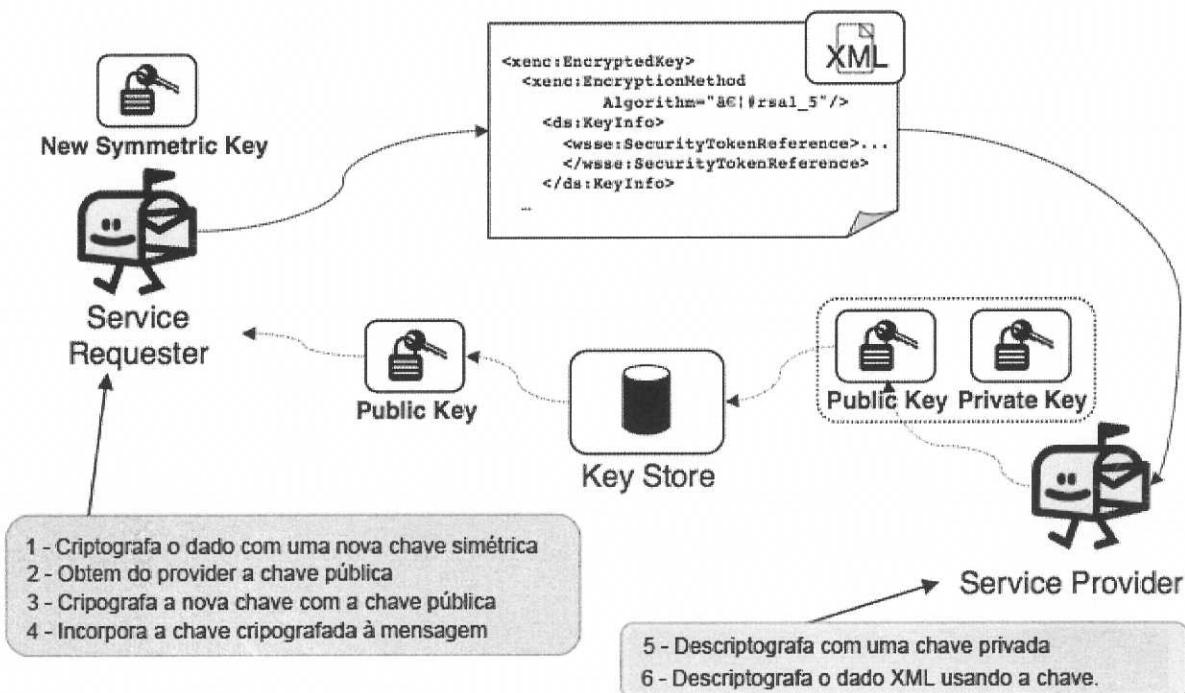
- O atributo **ValueType** indica qual o token de segurança, por exemplo, um certificado X.509.
- O **EncodingType** nos diz como a segurança é codificada, por exemplo, **Base64Binary**.

```
<wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext">
  <wsse:BinarySecurityToken
    Id="X509Token"
    xmlns:wsse=http://schemas.xmlsoap.org/ws/2002/12/secext
    ValueType="wsse:X509v3"
    EncodingType="wsse:Base64Binary">
      MIIEZzCCA9CgAwIBAgIQEmtJZc0...
  </wsse:BinarySecurityToken>
  ...
</wsse:Security>
```

A seguir, um exemplo completo utilizando um certificado X.509:

```
<wsse:Security
  xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext">
  <wsse:BinarySecurityToken Id="X509Token"
    xmlns:wsse=http://schemas.xmlsoap.org/ws/2002/12/secext
    ValueType="wsse:X509v3"
    EncodingType="wsse:Base64Binary">
      MIIEZzCCA9CgAwIBAgIQEmtJZc0...
  </wsse:BinarySecurityToken>
  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo> ... </ds:SignedInfo>
    <ds:SignatureValue> ... </ds:SignatureValue>
    <ds:KeyInfo>
      <wsse:SecurityTokenReference>
        <wsse: Reference URI="#X509Token" />
      </wsse:SecurityTokenReference>
    </ds:KeyInfo>
  </ds:Signature>
```

15.5 Usando criptografia XML



A Critografia XML especifica o uso de um mecanismo de chave criptografada com RSA (public key). É necessário lembrar que uma critografia assimétrica, usa uma chave pública para criptografar e uma privada para descriptografar.

Então a chave deverá ser criada pelo sender e criptografada com uma chave pública recebida.

A chave criptografada é incluída na mensagem e o receiver descriptografa com sua chave privada, baseado no elemento **KeyInfo**.