CARLOS EDUARDO GESSER

UMA ABORDAGEM PARA A INTEGRAÇÃO DINÂMICA DE SERVIÇOS WEB EM PORTAIS

FLORIANÓPOLIS
2006

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CURSO DE PÓS-GRADUAÇÃO EM ENG. ELÉTRICA

UMA ABORDAGEM PARA A INTEGRAÇÃO DINÂMICA DE SERVIÇOS WEB EM PORTAIS

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a
obtenção do grau de Mestre em Engenharia Elétrica.

CARLOS EDUARDO GESSER

Florianópolis, agosto de 2006.

UMA ABORDAGEM PARA A INTEGRAÇÃO DINÂMICA DE SERVIÇOS WEB EM PORTAIS

Carlos Eduardo Gesser

'Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica, Área de Concentração em *Automação e Sistemas*, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina.'

Prof. Ricardo José Rabelo, Dr.
Orientador

Prof. Nelson Sadowski
Coordenador do Programa de Pós-Graduação em Engenharia Elétrica

Banca Examinadora:

Prof. Ricardo José Rabelo, Dr.
Presidente

Prof. Carlos Barros Montez, Dr.

Prof. Frank Augusto Siqueira, Dr.

Prof. Rômulo Silva de Oliveira, Dr.

AGRADECIMENTOS

Agradeço inicialmente a meus pais: Vilmar e Cida, que concordaram a minha maluquice de vir estudar em Florianópolis. Sem o apoio deles não teria nem sequer me formado na graduação, quanto mais estar entregando esta dissertação.

Seria impossível não agradecer a minha namorada Juliana, que me aturou durante toda a escrita deste trabalho. Consegui sobreviver ao *stress* que é a escrita de uma dissertação graças a seu grande amor e carinho por mim.

Devo muito ao grupo GSIGMA, que tem sido minha casa desde meados de 2000. Meu agradecimento principal vai para Ricardo Rabelo, o coordenador do grupo e meu orientador. Agradeço a Rui, Baldo e Leandro, meu colegas de laboratório desde o início. Entramos juntos aqui e temos muitas histórias pra contar. Agradeço também ao pessoal que foi entrando e saindo no meio do caminho: Saulo, Piazza e "Baiano". E é claro, agradecimentos especiais a Edmilson e Xanda também.

Finalmente, agradeço aos professores e funcionários da PGEEL.

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários para obtenção do grau de Mestre em Engenharia Elétrica.

UMA ABORDAGEM PARA A INTEGRAÇÃO DINÂMICA DE SERVIÇOS WEB EM PORTAIS

Carlos Eduardo Gesser

Agosto/2006

Orientador: Prof. Ricardo José Rabelo, Dr.
Área de Concentração: Automação e Sistemas
Palavras-chave: Serviços Web, Interfaces de Usuário, Portais, Portlets
Número de Páginas: 100

RESUMO: O desenvolvimento de aplicações baseadas no paradigma de Arquiteturas Orientadas a Serviços (ou SOA) vem tendo um grande e rápido crescimento nos últimos tempos como uma forma de reduzir custos e o tempo de desenvolvimento de *software*. Do ponto de vista de implementação, a tecnologia dos serviços *web* vem sendo a mais amplamente utilizada e adotada pelas principais empresas para seus sistemas de *software*. Serviços *web* provêem uma interface de invocação bem definida, mas apenas para outras aplicações. Para usuários, faz-se necessário o desenvolvimento de uma interface de usuário para cada serviço. Paralelamente a isto, cada vez mais as empresas utilizam portais como ambiente de interação usuário-aplicações, permitindo inclusive personalizações de acesso e visualização. Todavia, com as tecnologias atualmente disponíveis, o desenvolvimento e implantação de interfaces de usuário para serviços *web* em portais requerem por parte do programador grande conhecimento de uma série de tecnologias de informação, além de ser um processo relativamente lento e custoso. Esta dissertação propõe uma abordagem que automatiza esse processo. Ela permite que se gere dentro de um portal, de forma automática e dinâmica, a interface de usuário para um serviço baseado em seu descritor. Através desta, usuários podem fazer invocações ao serviço de forma dinâmica. Com vistas a analisar e avaliar a proposta, um componente genérico de portal foi implementado.

Abstract of Dissertation presented to UFSC as a partial fulfillment of the requirements for the degree of Master in Electrical Engineering.

AN APPROACH FOR THE DYNAMIC INTEGRATION OF WEB SERVICES IN PORTALS

Carlos Eduardo Gesser

August/2006

Advisor: Prof. Ricardo José Rabelo, Dr
Area of Concentration: Automation and Systems
Key words: Web Services, User Interfaces, Portals, Portlets
Number of Pages: 100

ABSTRACT: The development of applications based on the Service-Oriented Architecture (or SOA) paradigm is growing greatly and fastly nowadays as a way to reduce costs and time of software development. From the implementation point of view, the web services technology is clearly the most used and adopted one by the main companies for theis software systems. Web services have well defined invocation interfaces, but only for other applications. For end-users, it is rather necessary to develop a user interface for each service. Alongside this, portals have been more and more used by companies for user-applications interaction, also allowing personalization of access and visualization. However, using the current technologies, the development and deployment of user interfaces for web services in portals requires, from the programmer side, great knowledge of a good number of associated information technologies, besides being a relatively slow and costly process. This dissertation proposes an approach that automates this process. It allows the automatic and dynamic generation of the web service's user interface in a portal, based on the service descriptor. Through this interface, users can invoke services dynamically. In order to assess and to evaluate the proposal, a generic portal component has been implemented.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Acrônimos

AJAX *Asynchronous JavaScript and XML* (JavaScript Assíncrono e XML)

API *Application Programming Interface* (Interface para Programação de Aplicações)

CORBA *Common Object Request Broker Architeture* (Arquitetura Comum de Intermediário de Requisições de Objetos)

CVS *Concurrent Versions System* (Sistema de Versões Concorrentes)

DTD *Document Type Definition* (Definição de Tipo de Documento)

ERb *Embedded Ruby* (Ruby Embutido)

GNU GNU is Not Unix (GNU Não é Unix)

HTML *HyperText Markup Language* (Linguagem de Marcação de Hipertexto)

HTTP *Hypertext Transfer Protocol* (Protocolo de Tranferência de Hipertexto)

IDL *Interface Description Language* (Linguagem de Descrição de Interface)

IP *Internet Protocol* (Protocolo da Internet)

JSF *JavaServer Faces* (Faces de Servidor Java)

JSP *JavaServer Pages* (Páginas de Servidor Java)

LGPL *Library General Public License* (Licença Geral Pública para Bibliotecas)

MVC *Model-View-Controller* (Modelo-Visão-Controlador)

OKI *Open Knowledge Initiative* (Iniciativa de Conhecimento Aberto)

OSDI *Open Service Interface Definition* (Definição Aberta de Interface de Serviço)

QoS *Quality of Service* (Qualidade de Serviço)

PDF *Portable Document Format* (Formato de Documento Portável)

RPC *Remote Procedure Call* (Chamada Remota de Procedimento)

RSS *Really Simple Syndication* (Distribuição Realmente Simples de Conteúdo)

SLA *Service Level Agreement* (Acordo de Nível de Serviço)

SOA *Service-Oriented Architecture* (Arquitetura Orientada a Serviços)

UDDI *Universal Description, Discovery, and Integration* (Descrição, Descoberta e Integração Universais)

UML *Unified Modeling Language* (Linguagem de Modelagem Unificada)

URI *Universal Resource Identifier* (Identificador Universal de Recursos)

URL *Universal Resource Locator* (Localizador Universal de Recursos)

WAR *Web Archive* (Arquivo *Web*)

WML *Wireless Markup Language* (Linguagem de Marcação Sem-Fio)

WSDL *Web Services Description Language* (Linguagem de Descrição de Serviços *Web*)

WS-I *Web Services Interoperability* (Interoperabilidade de Serviços *Web*)

WSRP *Web Services for Remote Portlets* (Serviços *Web* para *Portlets* Remotos)

XHTML *eXtensible HyperText Markup Language* (Linguagem de Marcação de Hipertexto Extensível)

XML *eXtensible Markup Language* (Linguagem de Marcação Extensível)

XSL *eXtensible Stylesheet Language* (Linguagem Extensível de Folha de Estilo)

XSL-FO *XSL Formatting Objects* (Formatação de Objetos XSL)

XSLT *XSL Transformations* (Transformações XSL)

Capítulo 1

Introdução

1.1 Origem do Trabalho

Segundo BEA Systems (2004), a integração de aplicações é uma das questões mais críticas enfrentadas por gerentes de tecnologia da informação empresarial. As abordagens tradicionais de desenvolvimento e integração de aplicações são complexas, caras e inflexíveis, não suportando as necessidades das organizações atuais, que primam pela dinamicidade.

Jardim-Gonçalves et al. (2006) diz que um requisito básico para a integração de aplicações é o de que elas precisam ser preparadas com mecanismos adequados e interfaces abertas, na forma de serviços implementados seguindo os padrões efetivos. Isto leva ao desenvolvimento de *Arquiteturas Orientadas a Serviços* (SOAs).

Em uma SOA, todas as funções são definidas como serviços independentes com interfaces de invocação bem definidas, que podem ser chamadas em seqüências definidas para compor processos de negócios (Channabasavaiah et al., 2004).

SOAs podem ser implementada com várias tecnologias, sendo que atualmente a de serviços *web* vem sendo empregada pelas principais empresas de *software*[1]pois provê um meio padronizado para implementar SOAs a um custo razoável e utilizando métodos de comunicação largamente disponíveis nos dias de hoje (BEA Systems, 2004). De acordo com Huang e Chung (2003), o uso de serviços *web* tem sido visto como a chave para revolucionar a forma pela qual organizações fazem negócios. Como será visto em mais detalhes na seção 2.1, serviços *web* são aplicações auto contidas e modulares, que podem ser descritas, publicadas, localizadas e invocadas sobre a *Web*. A consultoria *Gartner* prevê 65% das grandes companhias americanas terão mais de um terço de seus sistemas baseados em SOAs até 2010, contra os 5% atuais (Cesar, 2006).

Em geral, usuários acessam serviços disponíveis em uma SOA através de um Portal. De acordo com BEA Systems (2004), um portal é um *site* da *web* que simplifica e personaliza o acesso a conteúdo, aplicações e processos. Eles estão ficando cada vez mais importantes para as empresas, que

[1]Entre elas: IBM, Microsoft, SAP, BEA e Oracle

tem cada vez mais a necessidade de prover para seus funcionários, parceiros e clientes uma visão integrada das aplicações, informações e processos de negócio. Portais permitem que suas funcionalidades sejam modularizadas na forma de *componentes de portal*.

Entretanto, segundo Hepper et al. (2005), a necessidade de se programar a camada de apresentação para cada serviço que se deseja agregar a um portal é um empecilho para as organizações. Apesar de serviços em uma SOA poderem ser dinamicamente ligados entre si para gerar os mais diversos tipos de aplicações, suas interfaces com o usuário ainda devem ser desenvolvidas da maneira tradicional, ou seja, programada de forma específica para cada serviço e suas operações. Isto implica em dificuldades, custo e tempo adicionais no desenvolvimento de aplicações. Esta dissertação de mestrado propõe um meio para minimizar consideravelmente este problema.

1.2 Problema de Pesquisa

Cada vez mais aplicações são implementadas com base em serviços *web*. Vassiliadis et al. (2006) prevêem que com o devido tempo, as companhias substituirão suas abordagens tradicionais de integração de informação pela prática orientada a serviços. Como já foi exposto, serviços possuem interfaces de invocação bem definidas, que permitem que aplicações, ou outros serviços, possam usá-los. Isto provê uma grande flexibilidade no processo de composição da aplicação.

Entretanto, para disponibilizar um serviço a um usuário esta flexibilidade não se faz presente. Usuários não podem fazer uso de um serviço a partir de sua interface de invocação. Ele precisa de uma interface de usuário para tanto.

O desenvolvimento de interfaces de usuário para invocar serviços web demanda que se crie a interface em si (com seus campos de entrada de dados para os parâmetros do serviço) e a lógica de programação para invocar o serviço com os parâmetros inseridos e informar ao usuário a resposta da invocação. Este é um processo tedioso e usualmente demorado e propenso a erros. Após isto, a interface gerada necessita ser implantada no portal, para que o serviço seja integrado. Esta proposta pressupõe que ganhos significativos no desenvolvimento de sistemas baseados em serviços *web* podem ser conseguidos se estes processos (desenvolvimento e implantação) puderem ser automatizados e efetuados dinamicamente.

O problema tratado por este trabalho é: como fazer para que as interfaces gráficas de usuário para serviços *web* possam ser geradas automática e dinamicamente em um componente de portal, de forma que estes serviços possam ser integrados e posteriormente invocados por usuários deste portal? A *integração* de um serviço em um portal é entendida como o processo que permite que usuários invoquem o serviço a partir do portal. Isto abrange a geração da interface de usuário e a invocação em si. O processo é considerado *automático* por não necessitar de intervenção humana (exceto no passo inicial da escolha do serviço). Finalmente o processo é dito *dinâmico* por acontecer com o servidor de portal em execução, sem a necessidade de compilação de código.

A interface de um serviço é uma descrição formal das operações, de seus respectivos parâmetros e de seus tipos de dados. A abordagem proposta parte do princípio que, através desta descrição, um

componente de portal possa gerar sua interface de usuário automática e dinamicamente para que, em um segundo passo, invocar o serviço usando os dados informados pelo usuário como parâmetros.

1.3 Objetivos do Trabalho

Os objetivos deste trabalho estão divididos em objetivos gerais e específicos.

1.3.1 Objetivo Geral

A contribuição científica deste trabalho, delineada por seu objetivo geral, é propor uma abordagem que permita que, a partir de uma descrição de serviço *web*, seja automática e dinamicamente construída uma interface gráfica de usuário, para que a partir dela seja possível a invocação dinâmica do referido serviço.

Feito isto, será desenvolvido um protótipo, na forma de um componente de portal, para permitir que serviços *web* possam ser integrados em portais, sem a necessidade de se programar um componente diferente para cada serviço diferente.

1.3.2 Objetivos Específicos

1. Concepção de um método para a geração automática e dinâmica de interfaces gráficas de usuário para serviços *web*.
2. Concepção de um método para a invocação dinâmica de um serviço *web* a partir da interface de usuário.
3. Implementação dos métodos e avaliação dos resultados.

1.4 Justificativa

Este trabalho justifica-se pelo ganho de produtividade que pode ser atingido utilizando-se o método proposto, em comparação com a abordagem tradicional de integração de serviços *web* em portais. A figura 1.1 os passos entre a publicação de um serviço *web* e seu acesso por um usuário de portal de acordo com as abordagens tradicionais. A descrição dos passos é apresentada a seguir.

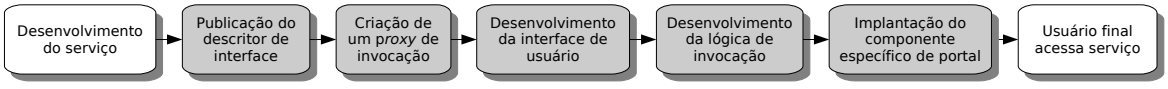

Figura 1.1: Abordagem tradicional para a integração de serviços em portais

- Tradicionalmente, para se acessar um serviço é necessária a criação de um *proxy* de invocação, em uma linguagem de programação específica. Há diversas ferramentas para serviços *web* que geram *proxies* para as mais variadas linguagens.

- A criação da interface é específica para cada serviço. Diferentes serviços necessitam de diferentes parâmetros de entrada, e possuem diferentes dados de saída, os quais devem ser apresentados aos usuários de formas específicas. Para portais *web* as interfaces de usuário são geralmente desenvolvidas em linguagens de marcação, como HTML.

- A lógica de invocação é a ligação entre a interface de usuário e o *proxy* de invocação em um componente de portal. Parâmetros de invocação inseridos pelo usuário devem ser passados ao *proxy*, que efetua a invocação do serviço e devolve a resposta. Esta resposta deve ser processada e apresentada ao usuário. A lógica deve ser específica para cada serviço.

- A implantação do componente gerado no portal implica em instalá-lo e configurar o portal para que o componente esteja acessível para seus usuários.

A abordagem proposta por este trabalho, e apresentada no capítulo 3, visa diminuir o esforço necessário para a integração de serviços *web* em portais. A necessidade de programação nos diversos passos apresentados anteriormente é eliminada com a introdução de um *Componente Genérico de Portal*, cuja interface de usuário adapta-se de acordo com os parâmetros do serviço *web* que se deseja integrar ao portal.

A figura 1.2 tenta ilustrar como a abordagem pretende ser aplicada. Partindo-se de um portal, um serviço de um dado repositório é integrado a ele, com a interface de usuário para invocá-lo gerada automaticamente. Este processo ocorre também dinamicamente, ou seja, não é preciso parar o *software* do portal, nem compilar código. Uma vez gerada a interface de usuário, invocações ao serviços podem ser feitas.

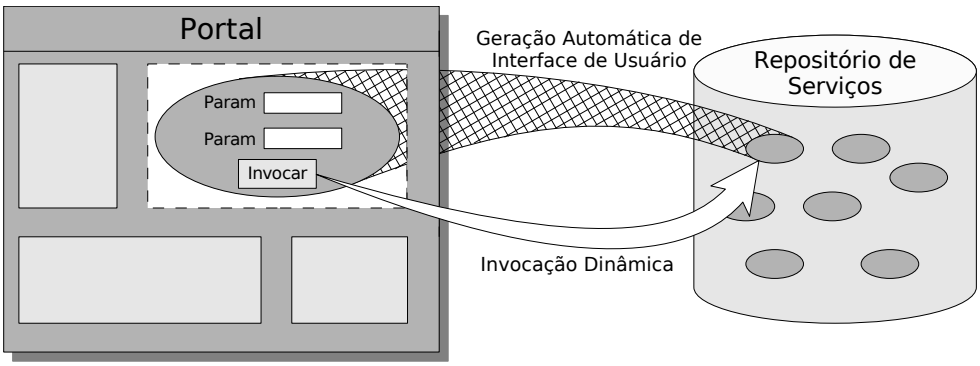

Figura 1.2: Integração de um Serviço em um Portal

1.5 Metodologia

1.5.1 Classificação da Pesquisa

Segundo a classificação proposta por da Silva e Menezes (2005), esta dissertação se enquadra da seguinte forma:

- **Quanto à Natureza da Pesquisa:** Aplicada, pois seu objetivo tem aplicação prática definida, que é integração automática e dinâmica de serviços *web* em portais.
- **Quanto à Forma de Abordagem do Problema:** Qualitativa, uma vez que não se aplica o uso de técnicas quantitativas (estatísticas) para classificação e análise da abordagem proposta.
- **Quanto aos Objetivos da Pesquisa:** Exploratória, já que ela consiste basicamente em uma pesquisa bibliográfica seguida da proposta de uma abordagem, sua implementação e avaliação como um protótipo.
- **Quanto aos Procedimentos Técnicos:** Predominantemente bibliográfica, sendo elaborada a partir de material já publicado, constituído principalmente de livros, artigos de periódicos e de conferências e normas de padronização de tecnologias.

1.5.2 Roteiro da Pesquisa

A partir da classificação previamente apresentada, a pesquisa que culminou com a escrita deste trabalho seguiu os seguintes passos:

1. Revisão bibliográfica da literatura relacionada, incluindo livros, artigos de conferências e de periódicos e especificações. As informações obtidas estão condensadas no capítulo 2, e na seção 3.3.
    - Os *sites* de indexação de artigos científicos foram pesquisados
        - ACM Portal – `http://portal.acm.org`
        - Cite Seer – `http://citeseer.ist.psu.edu`
        - IEEE Xplore – `http://ieeexplore.ieee.org`
        - Science Direct – `http://www.sciencedirect.com`
    - Os seguintes anais foram pesquisados
        - 2ª Conferência Internacional sobre Computação Orientada a Serviços
        - 7ª Conferência Internacional sobre Comércio Eletrônico
        - 24ª Conferência Internacional sobre Engenharia de *Software*
        - Conferência Internacional sobre Tecnologia da Informação: Pesquisa e Educação 2003

  - *Workshop* sobre *Design* e Evolução de *Software* Autônomo 2005
- Os seguintes periódicos foram pesquisados
  - *Computers in Industry*
  - *Computer Networks*
  - *Electronic Commerce Research and Applications*
  - *Journal of Object-Oriented Programming*
  - *Journal of Web Semantics*
  - *Proceedings of the IEEE*
  - *Telematics and Informatics*

2. Estudo de tecnologias: XML, XML *Schema*, Serviços *Web*, SOAP, WSDL, UDDI, *Portlets*, XSL, JSP, HTML.
3. Estudo detalhado das seguintes especificações:
   (a) Descritor de serviços *web*, para descobrir meios para a obtenção de informações para a criação dinâmica da interface de usuário.
   (b) Protocolo de serviços *web*, para desenvolver formas de se invocar dinamicamente um serviço.
   (c) Registros remotos de serviços *web*, para a pesquisa dinâmica de serviços nestes repositórios.
   (d) Componentes de portais, para a implementação do protótipo.
4. Avaliação do protótipo, através de testes com diversos serviços *web*.
5. Escrita da dissertação, com base nos dados obtidos anteriormente.

1.6 Estrutura do Trabalho

Este trabalho é estruturado da seguinte forma.

O capítulo 2 é formado por um resumo detalhado das tecnologias e padrões atuais de serviços *web* e criação interfaces de usuário respectivamente.

O capítulo 3 apresenta a abordagem proposta, que inclui os algoritmos e estruturas para a geração dinâmica de interfaces de usuário em componentes de portais a partir de descrições de serviços e para a invocação dinâmica destes. Também é apresentada a revisão do estado-da-arte e de trabalhos relacionados, em comparação com a abordagem proposta.

O capítulo 4 detalha a implementação do protótipo, sua avaliação e os resultados obtidos.

O capítulo 5 mostra as conclusões finais do trabalho, e que trabalhos futuros podem derivar dele.

Capítulo 2

Tecnologias

2.1 Tecnologias para Serviços Web

De acordo com Haas e Brown (2004), um serviço *web* é um sistema de software concebido para suportar interações entre máquinas em uma rede, de forma interoperável. Ele possui uma interface descrita em uma linguagem passível de processamento por computadores – WSDL. Outros sistemas interagem com um serviços web através de mensagens SOAP[1], tipicamente transportadas sobre HTTP.

Wang et al. (2005) cita que o arcabouço de serviços *web* é dividido em três áreas: protocolos de comunicação, descrições de serviços, e descoberta de serviços. Para cada uma destas áreas há uma tecnologia associada, respectivamente:

- **SOAP:** Protocolo de comunicação entre serviços *web* e seus clientes.
- **WSDL:** Linguagem que provê um mecanismo para descrever serviços *web* formal e que computadores conseguem processar.
- **UDDI:** Diretório de registro de serviços *web*.

A base destas três tecnologias é a linguagem XML, que é abordada na seção 2.1.2. O protocolo SOAP é detalhado na seção 2.1.4. WSDL utiliza *XML Schema* para especificar a estrutura e o tipo dos dados que trafegam nas mensagens. A primeira é mostrada na seção 2.1.5, e a segunda na seção 2.1.3. Por fim, repositórios UDDI são acessados por mensagens SOAP, e são detalhados na seção 2.1.6.

2.1.1 Arquiteturas Orientadas a Serviços

Como apresentado na seção 1.1, uma das principais aplicações de serviços *web* é a implementação de SOAs, pois são a tecnologia que melhor atende os requisitos para tal. Estes requisitos, segundo

[1] Inicialmente SOAP era um acônimo para *Simple Object Access Protocol* (Protocolo Simples para Acesso a Objetos), mas em versões mais recentes passou a ser apenas um nome, pois o acrônimo foi considerado incorreto.

Singh e Huhns (2005), são:

- **Baixo acoplamento:** Os serviços da arquitetura não devem ter uma dependência forte entre si.
- **Independência de implementação:** Não se deve depender de características específicas de linguagens de programação ou ambientes de execução.
- **Configuração flexível:** Os diferentes serviços devem poder ser ligados entre si de forma dinâmica e configurável.
- **Tempo de vida longo:** Os serviços devem existir por tempo suficiente para que sejam descobertos e utilizados até se obter confiança em seu comportamento.
- **Granularidade:** As funcionalidades de um sistema devem ser divididas em vários serviços.
- **Distribuição:** Os serviços devem ficar distribuídos, para aproveitar melhor os recursos computacionais.

Segundo Agrawal et al. (2005), SOA combina a habilidade de invocar objetos remotos e funções com ferramentas para a descoberta dinâmica de serviços, com ênfase na interoperabilidade. Entretanto, de acordo com MacKenzie et al. (2006), SOA não é, por si só, uma solução para problemas para um dado domínio, e sim um paradigma de organização e entrega que permite obter mais valor das competências localmente disponíveis e daquelas controladas por terceiros.

2.1.1.1 Modelo de referência de SOA

MacKenzie et al. (2006) apresenta um arcabouço abstrato para o entendimento das entidades relevantes e seus relacionamentos dentro de uma SOA, bem como para o desenvolvimento de padrões consistentes neste ambiente. Este modelo de referência foi desenvolvido com o propósito de unificar e padronizar a definição de SOA. Os seguintes conceitos são especificados:

- **Serviço**: Mecanismo que permite o acesso a uma ou mais capacidades, com acesso provido de acordo com uma interface pré-estabelecida e exercido consistentemente com as restrições e políticas especificadas na sua descrição.
- **Descrição de Serviço**: Informação necessária para usar ou avaliar o uso de um serviço. Seu propósito é facilitar a interação e a visibilidade.
- **Interação**: Série de troca de informações para a invocação de uma ação, resultando em uma efeito no mundo real.
- **Visibilidade**: Aptidão daqueles com capacidades que permite que aqueles com necessidades interajam consigo.
- **Efeito no Mundo Real**: O efeito real do uso de um serviço, podendo consistir do recebimento de uma informação ou de uma mudança de estado de alguma entidade envolvida.

2.1.2 XML

A XML (*eXtensible Markup Language*, Linguagem de Marcação Extensível) é uma linguagem de marcação de propósito geral, capaz de representar diferentes tipos de dados, que é usada para a criação de linguagens de marcação para um propósito específico, também chamadas de dialetos.

Uma linguagem de marcação é uma linguagem que combina texto e informações extra sobre este. Assim, qualquer trecho de texto pode conter informações extra, que podem informar desde a forma como o trecho deve ser apresentado (tamanho, cor, estilo, etc.) como seu propósito semântico.

A especificação *Extensible Markup Language 1.0* (Bray et al., 2004) define a sintaxe da linguagem, que será mostrada a seguir, que define uma forma de se criar linguagens específicas, através de DTD (*Document Type Definition*, Definição de Tipo de Documento). Apesar de fazer parte da especificação, DTD se tornou um tanto quanto obsoleto com a introdução de *XML Schema*, que será apresentado na seção 2.1.3.

2.1.2.1 Sintaxe

A unidade da linguagem XML é o *documento*, cujos principais constituintes são *elementos* e *atributos*. A figura 2.1 contém um exemplo de documento XML que apresenta estes objetos, e ilustra as formas sob as quais eles podem se apresentar.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<mensagem remetente="Carlos" data="2006-04-22">
  <destinatario nome="Eduardo" />
  <assunto>Exemplo de documento XML</assunto>
  <corpo>
    Este é um exemplo simples do documento XML, que mostra os principais
    elementos de sua <destaque>sintáxe</destaque>, que serão apresentados
    a seguir.
  </corpo>
</mensagem>
```

Figura 2.1: Exemplo de Documento XML

Um documento XML é constituído essencialmente por *elementos*. Elementos são definidos por *tags* que indicam o início e o fim de seu escopo. A *tag* inicial de um elemento pode contem *atributos*, na forma de pares de nome e valor. Entre os *tags* inicial e final, um elemento pode conter texto ou outros elementos. Elementos que contenham apenas texto são chamados de *elementos simples*, os demais (que contêm atributos ou outros elementos aninhados) são chamados de *elementos complexos*, sendo que elementos aninhados são chamados de seus *elementos-filhos*.

Um, e apenas um, elemento deve ser declarado no escopo raiz (exceção feita a elementos de cabeçalho). Todos os demais elementos devem ocorrer no escopo de outros elementos. Um documento XML representa desta forma uma estrutura de dados hierárquica. Comentários podem ser inseridos em documentos, e são delimitados por `<!-` e `-->`.

2.1.2.2 Esquemas de Validação

A sintaxe de documentos XML pode ser validada em dois níveis. Primeiramente, se estiver de acordo com a sintaxe descrita anteriormente (e detalhada formalmente na especificação), o documento é dito *bem formado*. Um segundo nível de validação sintático é possível com o uso de esquemas de validação, que delimitam a forma como elementos podem apresentar-se em um documento. Essas delimitações ocorrem em geral de duas formas:

- Na estrutura do documento, isto é, quais elementos e atributos podem fazer parte documento, e em qual estrutura hierárquica.
- Na forma como o texto contido em elementos e atributos deve ser interpretado [2].

Quando um documento está em conformidade com um esquema, o documento é dito *válido* de acordo com o referido esquema. Uma linguagem para definição de esquemas é a DTD, que faz parte da especificação XML. Outra é a *XML Schema*, que será detalhado a seguir. Para um aprofundamento sobre XML e DTD, pode-se consultar o trabalho de Tramontin Jr. (2004) e o padrão XML W3C (Bray et al., 2004).

2.1.2.3 Namespaces

É comum documentos XML terem elementos definidos por diferentes esquemas. Como exemplo, documentos WSDL, que descrevem serviços *web* (e serão detalhados na seção 2.1.5), são definidos por um esquema, e os tipos de dados que são usados nas mensagens dos serviços são definidos por outro(s). Para evitar choques de nome em elementos e atributos nestes casos, foram introduzidos os *namespaces*, através da especificação *Namespaces in XML* (Bray et al., 1999).

Namespaces são associados a esquemas de validação. Documentos que façam uso de esquemas com *namespaces* associados devem associar um prefixo a cada um destes *namespaces*, Elementos e atributos que tenham sido declarados sob um determinado *namespace* devem ter o respectivo prefixo. Um exemplo é apresentado na figura 2.2

Um *namespace* é atribuído a um prefixo através de um atributo especial, que usa o prefixo *xmlns*, como destacado na figura. Um *namespace* é um URI (*Universal Resource Identifier*, Identificador Universal de Recursos), podendo ser um URL. Mas estes URLs não precisam apontar para algum recurso, basta que sejam um identificador único.

2.1.3 XML Schema

XML Schema é uma linguagem baseada em XML, para a criação de esquemas de validação para XML, descrita pela especificação *XML Schema* (Thompson et al., 2004). Um exemplo de *XML*

[2] Por exemplo, se o texto representa uma data, ou um dado numérico

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<m:mensagem xmlns:m="http://www.gsigma.ufsc.br/mensagens"
     m:remetente="Carlos" m:data="2006-04-22">
  <m:destinatario m:nome="Eduardo" />
  <m:assunto>Exemplo de documento XML com Namespaces</m:assunto>
  <m:corpo xmlns:fm="urn://formatos-texto">
    Neste exemplo de documento XML, elementos e atributos pertencem a
    <fm:destaque>namespaces</fm:destaque>. Os elementos de formação
    de texto presentes no corpo da mensagem são de um
    <fm:destaque>namespace</fm:destaque> diferente dos demais.
  </m:corpo>
</m:mensagem>
```

Figura 2.2: Exemplo de documento XML usando namespaces

Schema, que será utilizado para demonstrar suas principais características, é apresentado na figura 2.3.

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

<xs:element name="mensagem">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="msg:destinatario" maxOccurs="unbounded"/>
      <xs:element name="assunto" type="xs:string"/>
      <xs:element name="corpo" type="msg:tipoCorpo"/>
    </xs:sequence>
    <xs:attribute name="remetente" type="xs:string" use="required"/>
    <xs:attribute name="data" type="xs:date"/>
 </xs:complexType>
</xs:element>

<xs:element name="destinatario">
  <xs:complexType>
    <xs:attribute name="nome" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<xs:complexType name="tipoCorpo">
  <xs:sequence>
    <xs:any />
  </xs:sequence>
</xs:complexType>
```

Figura 2.3: Exemplo de XML Schema

Um documento *XML Schema* é constituído por declarações de elementos, atributos e tipos. A cada elemento ou atributo é associado um tipo, o que pode ser feito de duas maneiras:

- Declarando-se o tipo juntamente com o elemento, como no caso dos elementos *mensagem* e *destinatario*.
- Referenciando-se um tipo declarado separadamente, através do atributo *type*. O tipo pode ser declarado no mesmo documento, como *tipoCorpo*, ou importado de um outro *XML Schema*, como os tipos *xs:string* e *xs:date*.

Tipos, tanto declarados juntamente a elementos como desvinculadamente, podem ser de duas categorias:

- **Tipos Simples:** São os tipos associados a atributos e elementos simples. Representam um conjunto de restrições aplicadas a uma seqüência de caracteres, isto é, delimitam que tipo de dado o elemento ou atributo correspondente poderá conter.
- **Tipos Complexos:** São os tipos associados a elementos complexos. Tipos complexos delimitam quais atributos um elemento pode ter, assim como quais elementos filhos poderão aparecer, em qual ordem e quantidade, e se texto entre elementos filhos será permitido.

2.1.3.1 Tipos Simples

Os tipos simples utilizados no exemplo da figura 2.3 (*string* e *date*) são definidos pela especificação. A tabela 2.1 mostra os tipos simples padrão mais utilizados.

Tipo	Exemplo	Descrição
string	Alô Mundo	Texto
integer	42	Números inteiros
decimal	15.64	Números com ponto decimal
hexBinary	C1FF42D4	Números codificados em hexadecimal
boolean	true/false 1/0	Booleanos, na forma literal ou numérica
date	2006-03-06	Representa uma data do ano
time	17:42:16.000	Representa uma hora do dia

Tabela 2.1: Alguns tipos pré-definidos

Em geral, estes tipos são suficientes para os mais diversos fins. Para os demais casos pode-se criar novos tipos simples, a partir de tipos simples existentes, através de derivação por restrição. Neste processo, o novo tipo aceita um subconjunto dos valores que o tipo original aceita

Diversas restrições podem ser impostas a um tipo simples para se criar um novo tipo. Entre pode-se destacar:

- Restrição de faixa de valores para tipos numéricos.
- Enumeração de valores aceitos.

- Uso de expressões regulares.
- Restrição do número de caracteres mínimo e/ou máximo.

A figura 2.4 demonstrada declarações de tipos simples, cada uma fazendo uso de uma das restrições previamente citadas.

```
<xs:simpleType name="byte">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="255"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="genero">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Masculino"/>
    <xs:enumeration value="Feminino"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="cep">
  <xs:restriction base="xs:string">
    <xs:pattern value="[0-9]{5}-[0-9]{3}"/>
  </xs:restriction>
</xs:simpleType>

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

Figura 2.4: Tipos simples derivados por restrição

O primeiro tipo simples declarado – *byte* – é baseado no tipo *integer*, mas cujos valores deverão estar entre 0 e 255. O tipo *genero* é derivado a partir do tipo *string*, com a restrição de apenas aceitar os valores *Masculino* e *Feminino*. Tipos com esta forma de restrição são chamados de *tipos enumerados*. O tipo *cep* também é derivado de *string*, mas é limitado por uma expressão regular, que diz que ele deve ser formado por cinco dígitos, seguido de um hífen, seguindo de mais três dígitos. Por fim, *password* é uma *string* com no mínimo cinco e no máximo 8 caracteres.

2.1.3.2 Tipos Complexos

Tipos complexos podem ser criados de três formas:

- Pela enumeração de seus elementos e atributos.

- Pela restrição de um tipo complexo existente.
- Pela extensão de um tipo simples ou complexo existente.

A figura 2.5 mostra a declaração de tipos complexos.

```
<xs:complexType name="pessoa">
  <xs:all>
    <xs:element name="nome" type="xs:string" />
    <xs:element name="sobrenome" type="xs:string" />
    <xs:element ref="endereço" />
  </xs:all>
  <xs:attribute name="idade" type="xs:integer" />
</xs:complexType>

<xs:element name="endereço">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="rua" type="xs:string" />
      <xs:element name="numero" type="xs:integer" />
      <xs:element name="bairro" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Figura 2.5: Declaração de tipo complexo

Neste exemplo, é criado um tipo complexo chamado *pessoa*. Ele é composto por três elementos: *nome* e *sobrenome*, do tipo *string* e *endereço*, que referencia um elemento declarado a seguir, além do atributo *idade*, que é do tipo *integer*.

Elementos declarados devem ser englobados por um *agrupador*. Existem três tipos de agrupador:

- **sequence:** Os elementos devem aparecer na ordem que foram declarados.
- **all:** Os elementos podem aparecer em qualquer ordem.
- **choice:** Apenas um dos elementos pode aparecer.

Pode-se indicar a cardinalidade de cada elemento, isto é, o número mínimo e máximo de ocorrências que ele pode apresentar, através dos atributos *minOccurs* e *maxOccurs*. O primeiro indica o número mínimo de vezes que o elemento pode ser repetido, o segundo indica o número máximo de vezes. Para *maxOccurs* pode-se usar um valor numérico, ou o valor especial *unbounded*, que especifica que o número máximo de repetições é ilimitado. Um exemplo é mostrado na figura 2.6

No exemplo acima, o elemento *a* é opcional, pois pode aparecer uma vez ou nenhuma. Já *b* é obrigatório, pois pode aparecer no máximo e no mínimo uma vez. O elemento *c* pode aparecer de duas a cinco vezes. E, *d* e *e* não tem limite máximo de aparecimentos, mas o primeiro deve aparecer pelo menos uma vez, enquanto o segundo pode não aparecer por completo. Na omissão de qualquer

```
<xs:complexType name="teste">
  <xs:sequence>
    <xs:element name="a" type="xs:string" minOccurs="0" maxOccurs="1"/>
    <xs:element name="b" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="c" type="xs:string" minOccurs="2" maxOccurs="5"/>
    <xs:element name="d" type="xs:string" minOccurs="1" maxOccurs="unbounded"/>
    <xs:element name="e" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Figura 2.6: Possibilidades de uso dos atributos minOccurs e maxOccurs

um destes atributos, é assumido que seu valor é *1*, o que significa que elementos são, por padrão, obrigatórios.

Atributos são declarados sempre após os elementos e, como sua ordem em documentos XML não é relevante, eles não são declarados dentro de agrupadores. Atributos podem ser de uso obrigatório ou opcional. Isto é controlado pelo atributo *use*, que pode receber os valores *required* (obrigatório) e *optional* (opcional). Na omissão deste atributo, é assumido que o atributo é opcional.

2.1.4 SOAP

Como dito pela especificação SOAP (Gudgin et al., 2003), SOAP é um paradigma de troca de mensagens em mão única e sem manter estado, mas que permite que aplicações façam interações mais complexas a partir dele combinando essas mensagens.

De acordo com Singh e Huhns (2005), a intenção original do SOAP era prover serviços de chamada remota de procedimento (RPC) escritos em XML, mas atualmente ele funciona como um protocolo simples e leve para a troca de mensagens XML, executado sobre protocolos de troca de texto da Web.

2.1.4.1 Estrutura das mensagens

A figura 2.7 mostra a estrutura hierárquica de uma mensagem SOAP. Mensagens SOAP são chamadas de *Envelopes SOAP*, que são constituídos por um cabeçalho e um corpo, cujos elementos XML correspondentes se chamam *Header* e *Body* respectivamente.

O elemento *Header* é opcional. Seus elementos filhos são *elementos de cabeçalho*, cujas informações não fazem parte da *carga útil* do envelope. Seu conteúdo são informações de controle, como diretivas ou informações contextuais, que diferentes nós no caminho entre o emissor e o receptor da mensagem podem inspecionar, adicionar ou remover.

O elemento *Body* é obrigatório. Seus elementos filhos são a *carga útil* da mensagem, cujo conteúdo pode representar uma RPC, ou ser um conjunto de elementos arbitrários. Além de elementos de

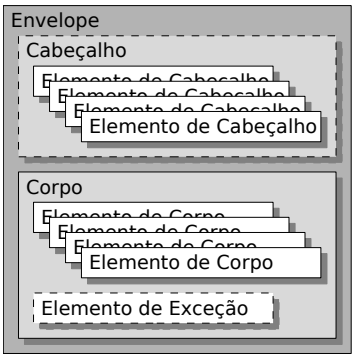

Figura 2.7: Estrutura de uma mensagem SOAP

corpo, o elemento *Body* pode conter um elemento de exceção (*Fault*), quando a mensagem representa uma resposta problemática, contendo informações sobre o problema.

As figuras 2.8 e 2.9 mostram, respectivamente, uma requisição e uma resposta SOAP, juntamente com os cabeçalhos HTTP.

```
POST /ws/serviços HTTP/1.1
Host: www.exemplo.com
Content-Type: application/soap+xml; charset=iso-8859-1
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <obterInfoProduto xmlns="http://www.exemplo.com/ws">
      <id>827635</id>
    </obterInfoProduto>
  </soap:Body>
</soap:Envelope>
```

Figura 2.8: Requisição SOAP

2.1.4.2 Encaminhamento de mensagens

Como dito anteriormente, uma mensagem pode passar por nós intermediários antes de chegar ao seu destino final. Três tipos de nós existem:

- Emissor
- Intermediário
- Receptor Final

O corpo da mensagem destina-se ao receptor final, enquanto que os conteúdos de cabeçalho podem se destinar a nós intermediários ou mesmo ao receptor final.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=iso-8859-1
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <obterInfoProdutoResponse xmlns="http://www.exemplo.com/ws">
      <info-produto>
        <nome>Sofá 3 Lugares Amgisg</nome>
        <id>827635</id>
        <descricao>Sofá de 3 Lugares em Couro Preto.</descricao>
        <preco>949.99</preco>
        <disponivel>true</disponivel>
      </info-produto>
    </obterInfoProdutoResponse>
  </soap:Body>
</soap:Envelope>
```

Figura 2.9: Resposta SOAP

Elementos de cabeçalho podem conter atributos predefinidos para indicar como um nó deve reagir em relação a ele. Estes atributos são:

- ***role*:** Indica que o atributo é destinado a nós que desempenhem o papel especificado pelo atributo.
- ***mustUnderstand*:** Se presente, e com o valor *true*, o nó atual é obrigado a processar o elemento de cabeçalho.
- ***relay*:** Se presente, com o valor *true*, e o elemento de cabeçalho não for processado, ele deve ser repassado para o próximo nó.

2.1.4.3 Codificação SOAP

A especificação SOAP recomenda uma codificação especial para o conteúdo de mensagens, especialmente para aquelas que representem requisições ou respostas de RPC.

A codificação SOAP é baseada nas estruturas de dados de linguagens de programação. Os tipos de dados são divididos em duas categorias principais:

- **Tipos Escalares:** Tipos simples predefinidos do *XML Schema*. Contém apenas um valor.
- **Tipos Compostos:** Tipos construídos a partir de um conjunto de outros tipos (escalares ou compostos). Contém múltiplos valores.

Existem duas classes de tipos compostos:

- **Estruturas:** Valores são diferenciados uns dos outros por seus nomes. Cada valor tem um nome único.
- ***Arrays*:** Valores são diferenciados uns dos outros por sua posição apenas.

Elementos podem apresentar o atributo *nodeType*, contendo o nome do tipo associado, escalar ou composto. Arrays podem ainda ter os atributos *itemType* e *arraySize*, para indicar o tipo dos elementos do *array* e o número de elementos, respectivamente. A versão anterior do SOAP usava o atributo *arrayType* no lugar destes. Seu valor agrupava o tipo dos elementos e a dimensão do array na forma "`<tipo>[<dimensão>]`", como por exemplo: "`xs:string[100]`".

Em linguagens que suportam programação orientada a objetos, com estes sendo acessados por referências (ou ponteiros), não é incomum então um objeto ser referenciado mais de uma vez. Para simular este comportamento, a codificação SOAP faz uso dos atributos `ref` e `id`. Quanto um objeto for codificado para XML, a ele é adicionado o atributo `id`, cujo valor será um identificador único. Assim, referências a este objeto são feitas usando o atributo `ref`. Desta forma o mesmo objeto pode ser referenciado quantas vezes for necessário.

2.1.4.4 SOAP sobre HTTP

A especificação SOAP define apenas a sintaxe que as mensagens devem seguir, não se atrelando a nenhum protocolo de transporte. Um dos protocolos mais usados é o HTTP, protocolo padrão da *web* para a transferência de hipertexto. A especificação detalha esta combinação, mas qualquer protocolo de transferência de dados pode ser utilizado para a troca de mensagens SOAP.

A natureza de requisição/resposta do HTTP permite que mensagens SOAP sejam correlacionadas de forma simples, caso uma seja em resposta a outra. Isto é útil principalmente quando se usa SOAP para modelar RPC.

2.1.4.5 Perfil Básico WS-I

Segundo Singh e Huhns (2005), existem diversos pontos nas especificações para serviços web que são ambíguos, ou simplesmente coisas que podem ser feitas de maneiras diversas. Para minimizar estes problemas, a Organização para Interoperabilidade de Serviços *Web* WS-I juntou uma séries de recomendações sobre os padrões para serviços *web*, tentando minimizar estes problemas.

Estas recomendações foram publicadas como o *Perfil Básico WS-I* (Ballinger et al., 2004). Algumas dessas recomendações são o uso da versão 1.1 do SOAP, o uso de *XML Schema* para codificar as mensagens (em vez da codificação SOAP, descrita na seção 2.1.4.3), o uso de SOAP apenas sobre HTTP, com requisições *POST* e o uso de WSDL 1.1 para descrever os serviços.

2.1.5 WSDL

WSDL é uma linguagem derivada de XML para especificar a interface de um serviço web. De acordo com Chappell e Jewell (2002), WSDL se diferencia de outras linguagens de descrição de interfaces, como por exemplo IDL da OMG (OMG, 2002), pelo seu nível de extensibilidade, que permite:

- Descrever pontos de acesso e suas mensagens, independentemente do formato da mensagem ou do protocolo de rede usado.
- Tratar mensagens como descrições abstratas dos dados sendo trocados.
- Agrupar conjuntos de operações como um tipo abstrato, que pode depois ser mapeado para um protocolo e tipo de dados concreto.

WSDL é a pedra fundamental da arquitetura de serviços web, pois ela provê uma linguagem comum para descrever serviços e um meio para automaticamente integrá-los (Cerami, 2002).

A linguagem é uma especificação do W3C (Christensen et al., 2001), que está atualmente na versão 1.1. Uma versão 2.0 encontra-se em fase final de desenvolvimento. Este trabalho não a levará em conta.

2.1.5.1 Estrutura da WSDL

Um documento WSDL define um serviço como um conjunto de pontos de acesso à rede, ou *ports*. *Ports* e mensagens são definidos de forma abstrata, separadamente de suas instanciações para protocolos concretos que são chamadas de *bindings*.

Documentos WSDL tendem a ser bem extensos. A estrutura geral de um documento WSDL é apresentada na figura 2.10.

Os elementos com contorno tracejado são opcionais, os demais (com a exceção óbvia do elemento raiz) podem aparecer repetidamente.

Um serviço *web* é definido em WSDL pelas seguintes seções:

- ***definitions*:** É o elemento raiz de um documento WSDL.
- ***types*:** Contém definições de *tipos* de dados, de acordo com algum esquema.
- ***message*:** Define uma *mensagem*, que representa uma requisição ou resposta.
- ***portType*:** Agrupa mensagens em operações em uma interface abstrata, que na nomenclatura WSDL se chama *tipo de port*.
- ***binding*:** Liga um tipo de *port* a um protocolo e um formato de dados específicos.

Figura 2.10: Estrutura de um documento WSDL

- ***service*:** Declara um serviço, que é definido por *ports*, os quais são combinações de uma ligação com um endereço de rede.

Algumas seções podem conter *elementos de extensão*, que são elementos não especificados pela sintaxe padrão da WSDL. Estes elementos permitem que a linguagem se adapte a qualquer protocolo e formato de mensagens, bastando para isso que sejam definidos elementos de extensão específicos.

As próximas seções detalharão os elementos principais de um documento WSDL.

2.1.5.2 Tipos

Um documento WSDL pode conter opcionalmente um elemento *types*. Este elemento é destinado a conter definições de tipos, isto é, a estrutura dos dados que serão enviados nas mensagens.

A especificação não define nenhuma linguagem para a definição de tipos. É sugerido o uso de *XML Schema*, embora qualquer outra linguagem equivalente possa ser utilizada.

2.1.5.3 Mensagens

Mensagens são definidas pelo elemento *message*. Cada mensagem possui um nome e é composta por uma ou mais partes lógicas, definidas pelo elemento *part*. A ordem de declaração das partes de uma mensagem é irrelevante. Cada mensagem representa um fluxo de dados em sentido único sem especificar emissor ou receptor, apenas os dados que trafegam no fluxo.

Cada parte de mensagem possui um nome e um tipo de dados associado. Existem duas formas de se associar um tipo de dados a uma parte, apresentadas na figura 2.11.

```
<message name="enviarPedidoRequest">
    <part name="numero" type="xs:int"/>
    <part name="data" type="xs:date"/>
</message>

<message name="enviarPedidoResponse">
    <part name="result" element="tns:pedido"/>
</message>
```

Figura 2.11: As duas formas de declarar partes de mensagens

A primeira forma é vincular cada parte da mensagem a um tipo, e a segunda é associar cada parte a um elemento (usando os atributos *type* e *element* respectivamente). A diferença entre as duas abordagens é que, no primeiro caso, a parte em si representa o elemento, que terá o tipo indicado. No segundo, o elemento indicado será a parte da mensagem.

2.1.5.4 Tipos de Port

Um tipo de *port* é uma coleção de operações, compostas em geral por duas mensagens, uma de requisição e uma de resposta. Ele é definido pelo elemento *portType*, que contem uma coleção de operações, definidas pelo elemento *operation*. Operações são compostas por mensagens, que podem representar requisições, respostas ou falhas. Um exemplo de tipo de *port* é mostrado na figura 2.12.

```
<portType name="PedidoPortType">
    <operation name="enviarPedido" parameterOrder="numero data">
       <input message="tns:enviarPedidoRequest"/>
       <output message="tns:enviarPedidoResponse"/>
    </operation>
</portType>
```

Figura 2.12: Elemento *portType* com uma operação

Cada operação é composta normalmente por uma mensagem de requisição e uma de resposta, indicadas respectivamente pelos elementos *input* e *output*[3]. No caso de operações sujeitas a exceções, usa-se o elemento *fault* para indicar a mensagem que descreve a exceção.

[3] É possível criar operações *one way*, isto é, operações que não se espera resposta, suprimindo o elemento *output*.

Uma operação pode ainda conter o atributo *parameterOrder*. Este atributo é útil para serviços do tipo RPC, onde a ordem dos parâmetros de entrada é importante. O valor do atributo deve ser uma lista com os nomes das partes que compõem a mensagem de requisição, separados por um espaço em branco.

Pode-se traçar um paralelo entre as estruturas abstratas da WSDL e as de linguagens de programação, como mostrado na tabela 2.2.

WSDL	Linguagem de Programação
Tipo de *port*	Interface
Operação	Método
Parte de mensagem	Parâmetro

Tabela 2.2: Comparação entre estruturas da WSDL e de linguagens de programação

2.1.5.5 Ligações

Os tipos, mensagens e operações mostrados até agora são abstratos, sem nenhum vínculo com protocolos específicos. Este vínculo é definido pelo elemento *binding*, que liga um tipo de *port* a um protocolo e formato de dados específicos. Um exemplo de ligação com o protocolo SOAP para o tipo de *port* da figura 2.12 é mostrado na figura 2.13.

```
<binding name="PedidoSoapBinding" type="tns:PedidoPortType">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"/>
    style="rpc"
  <operation name="enviarPedido">
    <soap:operation soapAction="http://example.com/enviarPedido"/>
    <input>
      <soap:body use="encoded" namespace="http://example.com/ws"
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <soap:body use="encoded" namespace="http://example.com/ws"
         encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
</binding>
```

Figura 2.13: Exemplo de ligação SOAP

As indicações do protocolo e da forma de interpretar as descrições abstratas de dados são feitas por elementos de extensão, como destacado na figura.

O elemento *soap:binding*[4]vincula a ligação ao protocolo SOAP. O atributo *style* especifica o estilo das mensagens das operações nesta ligação. Existem duas possibilidades:

[4]O *namespace* definido pelo prefixo *soap* é: `http://schemas.xmlsoap.org/wsdl/soap/`.

- **rpc:** Mensagens estilo RPC, ou seja, seu conteúdo representa parâmetros e valores de retorno de uma chamada remota de procedimento.
- **document:** Mensagens contendo documentos quaisquer, sem a obrigação de representar uma RPC. Esta é a opção padrão, caso o atributo não esteja presente.

O elemento *soap:operation* aparece para cada operação do tipo de *port*. Ele pode ser utilizado para se especificar um estilo para cada operação (da mesma forma do estilo descrito anteriormente) e para se especificar a *ação SOAP*. A versão 1.1 do SOAP exigia que clientes HTTP enviem o cabeçalho *soapAction*, para indicar a operação a ser invocada. Esta exigência foi abolida na versão 1.2, já que esta informação já está presente no corpo da mensagem, e portanto é redundante.

Por fim, o elemento **soap:body** tem o propósito de especificar como as partes (*input* e *output*) de uma mensagem devem aparecer no corpo do envelope SOAP. O atributo *use* indica como as partes serão codificadas. Há duas possibilidades:

- ***literal*:** Os tipos abstratos devem ser interpretados como tipos concretos. Isto significa que o corpo terá exatamente a mesma estrutura especificada pelos esquemas declarados na seção *types*
- ***encoded*:** Os tipos abstratos devem ser interpretados de acordo com alguma codificação (indicada pelo atributo *encodingStyle*). Neste caso, é usado geralmente a codificação SOAP, descrita na seção 2.1.4.3

O estilo da operação (*rpc* ou *document*) e a codificação das mensagens (*literal* ou *encoded*) influenciam forma da criação dos envelopes SOAP.

2.1.5.6 Serviços

Serviços em WSDL são uma ligação que é tornada acessível. Serviços são definidos pelo elemento *service*, como mostrado na figura 2.14.

```
<service name="PedidoService">
    <port name="PedidoPort" binding="tns:PedidoBinding">
       <soap:address location="http://example.com/ws/pedidos"/>
    </port>
</service>
```

Figura 2.14: Elemento *service*

A associação de uma ligação a um endereço é feita pelos elementos *port*. O endereço é especificado por um elemento de extensão (aqui novamente usando SOAP), pois cada protocolo tem sua particularidade para endereçar serviços.

2.1.6 UDDI

O objetivo do da especificação do UDDI (*Universal Description, Discovery, and Integration*, Descrição, Descoberta e Integração Universais) (Clement et al., 2004) é a definição de um conjunto de serviços para suportar a descrição e a descoberta de:

1. Negócios, organizações e outros provedores de serviços *web*.
2. Os serviços *web* que estes disponibilizam.
3. As interfaces técnicas que podem ser utilizadas para acessar estes serviços.

UDDI provê uma infraestrutura fundacional interoperável para ambientes de *software* baseados em serviços *web*, tanto para serviços publicamente disponíveis como para serviços exposto internamente dentro de uma organização.

Segundo OASIS (2004), o protocolo UDDI é um elemento central do grupo de padrões que compõe a pilha de serviços *web*, e sua descrição define um método padronizado para publicar e descobrir os componentes de *software* baseado em rede de uma SOA.

2.1.6.1 Modelo de Dados

Os quatro principais elementos do modelo de dados UDDI, como descrito por Cerami (2002), são mostrados e relacionados na figura 2.15.

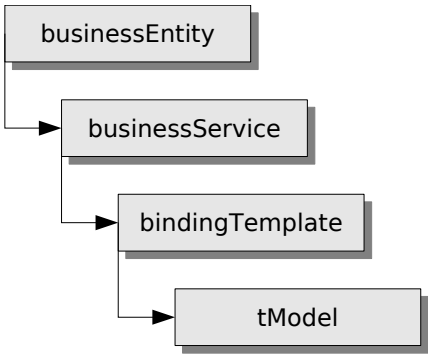

Figura 2.15: Relação entre as principais estruturas de dados do UDDI

- ***businessEntity*:** Contém informações sobre publicadores de serviços, como: nome, descrição, endereço e informações de contato.
- ***businessService*:** Inclui informações sobre um único serviço *web* (ou um grupo de serviços relacionados), como: nome e descrição.
- ***bindingTemplate*:** Possui informações sobre onde e como acessar um serviço *web* específico.

- ***tModel*:** É usado primariamente para prover referências externas de especificações técnicas sobre um serviço *web*, como por exemplo uma descrição WSDL. Também é utilizado para referenciar especificações de identificação e categorização.

A figura 2.16 apresenta um diagrama UML mais detalhado destas entidades.

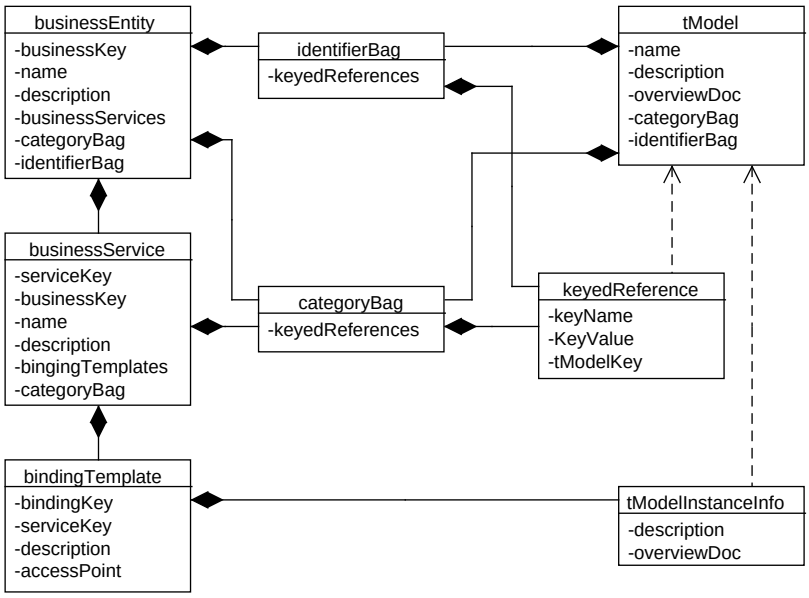

Figura 2.16: Modelo de dados UDDI

Os elementos *identityBag* e *categoryBag* permitem adicionar informações de identificação e de categorização respectivamente. Estas informações são inseridas na forma de elementos *keyedReference*. Estes elementos contêm um nome da respectiva identidade ou categoria, seu valor, e uma referência para um *tModel* com informações técnicas sobre a identidade ou categoria.

2.1.6.2 UDDI e WSDL

UDDI permite encontrar serviços *web*, e documentos WSDL permitem descrevê-los. Entretanto, segundo Hendricks et al. (2002), a especificação UDDI não menciona WSDL. A forma de se referenciar uma descrição WSDL a partir do UDDI é apresentada por Curbera et al. (2002). A descrição WSDL de um serviço *web* é uma especificação técnica, e portanto deve ser armazenada como um *tModel*, que contém o URL para a descrição como *overviewDoc*.

2.1.7 Conclusão sobre as Tecnologias para Serviços Web

Os serviços *web* empregam uma ampla quantidade de tecnologias, que são sumarizadas na figura 2.17.

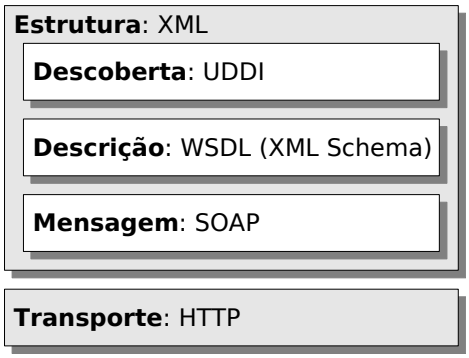

Figura 2.17: Pilha de Tecnologias para serviços *web*

XML é utilizado como formato de dados para toda a pilha de serviços *web*. Nesta pilha, mensagens são trocadas pelo protocolo SOAP, descritas pela linguagem WSDL (que faz internamente uso de *XML Schema*), e serviços podem ser descobertos por UDDI.

Todas estas tecnologias são padronizadas, e definem o que são serviços *web*. Portanto, seu estudo é essencial para a definição da abordagem proposta, a ser apresentado no capítulo 3.

2.2 Tecnologias para Construção de Aplicações na Web

As tecnologias apresentadas a seguir são utilizadas para o desenvolvimento de aplicações *web*, focadas na criação de interfaces de usuário. Inicialmente são apresentadas XHTML e XForms, duas linguagens de marcação complementares para a criação de interfaces de usuário. Em seguida a linguagem XSLT, que também pode ser utilizada para este fim. A seguir são apresentados *Servlets*, *Portlets*, tecnologias para a criação de aplicações *web* e componentes de portal, respectivamente. O padrão para acesso de *portlets* via serviços – WSRP – é mostrado logo após. Finalmente, é feita uma pequena resenha sobre a arquitetura MVC e arcabouços para a construção de aplicações *web* que usam desta arquitetura.

2.2.1 XHTML

XHTML (*eXtensible HyperText Markup Language*, ou Linguagem de Marcação de Hipertexto Extensível) é uma linguagem de marcação equivalente à linguagem HTML (da qual é considerada a sucessora), mas com um sintaxe mais restrita, por se basear em XML (Altheim e McCarron, 2001).

Por ter uma sintaxe restrita que a HTML, XHTML pode ser processada com o uso de menos recursos computacionais, o que torna esta linguagem apta para apresentar conteúdo em dispositivos de menor capacidade, como telefones celulares e *palm tops*. Além disto, ferramentas para tratar XML podem ser usadas com documentos XHTML, por serem linguagens relacionadas.

Segundo Musciano e Kennedy (2002), uma das principais vantagens da linguagem XHTML é que ela permite que novas construções sejam incluídas em documentos de maneira padronizada, em

namespaces próprios. Em HTML era comum navegadores introduzirem extensões próprias, incompatíveis com o padrão.

Um exemplo de documento XHTML é apresentado na figura 2.23.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
   "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" >
    <head>
        <title>Documento XHTML</title>
    </head>
    <body>
        <p>
            Este é um <em>exemplo</em> de<br/>
            Documento XHTML.
        </p>
    </body>
</html>
```

Figura 2.18: Exemplo de Documento XHTML

Atualmente está em desenvolvimento a versão 2 de XHTML, que possui muitas mudanças em relação à versão anterior. As mudanças visam simplificar a linguagem, com o intuito de delegar muitas de suas funcionalidades atuais a outras linguagens *XML*. Um exemplo disso é o uso de *XForms* para formulários, como será mostrado na próxima seção.

2.2.2 XForms

XForms é uma especificação (Boyer et al., 2006) de uma linguagem baseada em XML para representar modelos de dados e interfaces de usuário para interação com estes dados. Ela foi desenvolvida para substituir os formulários HTML tradicionais.

Raman (2003) descreve os princípios da linguagem XForms. Uma de suas características chave é a clara separação entre modelo de dados de sua apresentação ao usuário. O modelo de dados é formado por uma instância de documento XML que contém os dados inseridos pelo usuário, restrições usadas para validá-los, e meta dados sobre como eles devem ser enviados ao servidor *Web*. A apresentação ao usuário é feita a partir de componentes abstratos de alto nível para entrada de dados, vinculados a elementos do modelo. O uso deste tipo de componente permite que formulários XForms sejam apresentados da forma mais adequada ao dispositivo usado pelo usuário. Esta separação entre modelo de dados e sua visualização é feita com base no modelo MVC, que é apresentado na seção 2.2.6.

A figura 2.19 mostra um exemplo de modelo de dados XForms. Esta declaração é incluída dentro do cabeçalho (elemento `<head>`) XHTML. O atributo *schema* indica um documento *XML Schema* para validar os dados, indicando por exemplo que *idade* deve ser um número não negativo.

```
<model xmlns="http://www.w3.org/2002/xforms"
  schema="pessoa.xsd" id="p1">
  <instance>
    <pessoa xmlns="">
      <nome/>
      <idade/>
      <email/>
      <endereco> <rua/> <numero/> <cidade/> </endereco>
      <data-nasc> <dia/> <mes/> <ano/> </data-nasc>
    </pessoa>
  </instance>
</model>
```

Figura 2.19: Exemplo de Modelo XForms

Um exemplo de formulário de entrada de dados XForms é apresentado na figura 2.20. Os atributos destacados indicam a ligação entre o componente e o modelo. O elemento *label* indica o texto associado com o componente, *hint* contém uma mensagem de ajuda e *alert* uma mensagem para ser mostrada em caso de o dado inserido não ser válido. Estes componentes permitem ao usuário alterar elementos do modelo de dados, que é posteriormente enviado ao servidor. Apenas é especificado que este deve ser um campo de entrada de dados, o agente do usuário (navegador) é livre para implementá-lo da melhor forma possível, de acordo com o modelo de dados.

```
<input xmlns="http://www.w3.org/2002/xforms"
    xmlns:ev="http://www.w3.org/2001/xml-events" class="edit"
    model="p1" ref="/pessoa/idade">
  <label>Idade</label>
  <hint>Especifique sua idade</hint>
  <alert>A idade inserida, <output ref="/pessoa/idade"/>, não é válida!</alert>
</input>
```

Figura 2.20: Exemplo de Formulário XForms

Até o presente momento nenhum dos navegadores de internet mais usados suporta nativamente XForms, o que impossibilita seu uso prático em curto prazo.

2.2.3 XSLT

A XSL (*eXtensible Stylesheet Language*, Linguagem Extensível de Folha de Estilo) (Adler et al., 2001) é uma família de linguagens que permitem descrever como documentos XML devem ser formatados ou transformados. Três linguagens compõem a família:

- XSLT (*XSL Transformations*, Transformações XSL), linguagem usada para transformar documentos XML.
- XPath (*XML Path Language*, Linguagem de Caminhos XML) (Clark e DeRose, 1999), linguagem para endereçar partes de documentos XML.

- XSL-FO (*XSL Formatting Objects*, Formatação de Objetos XSL), linguagem para especificar a formatação visual de um documento XML.

A XSLT, descrita em detalhes em sua especificação por Clark (1999), permite descrever meio para que, a partir de um documento XML, seja gerado outro documento, em formato XML ou outros, como HTML, PDF e texto simples. Desta forma a transformação não destrói o documento original. Este processo é mostrado na figura 2.21.

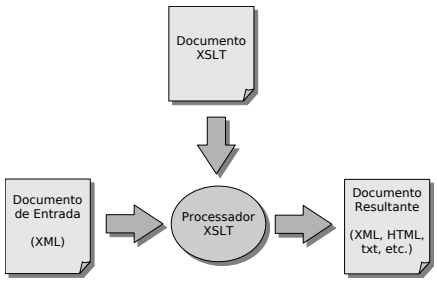

Figura 2.21: Processamento de Transformações XSL

XSLT é uma linguagem declarativa, no sentido que são especificadas regras indicando como transformar cada parte do documento original. Estas partes em cada regra são indicadas com expressões *XPath*.

A linguagem *XPath* (Clark e DeRose, 1999), permite a seleção de subconjuntos dos elementos de um documento XML. O processamento XSLT começa pelo elemento raiz do documento XML original. Quanto é encontrada uma regra que se adeqúe a ele, o conteúdo da regra é processado. Caso durante este processamento seja indicado que se deva processar outro conjunto de elementos, o processador tentará novamente encontrar a regra que melhor represente cada um dos elementos deste conjunto. O processamento continua desta forma até que nenhum processamento seja mais necessário. A figura 2.22 apresenta um documento XML, e a figura 2.23 um documento XSLT para transformá-lo em HTML.

```
<?xml version="1.0"?>
<pessoas>
  <pessoa nome="Carlos">
    <idade>25</idade>
    <cidade>Blumenau</cidade>
  </pessoa>
  <pessoa nome="Juliana">
    <idade>19</idade>
    <cidade>Florianópolis</cidade>
  </pessoa>
</pessoas>
```

Figura 2.22: Documento XML para transformação XSLT

Cada regra é definida por um elemento *template*, cujo atributo *match* contém uma expressão *XPath* para indicar quais os elementos XML que se qualificam para serem processados por esta regra.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="pessoas">
    <html>
      <body>
        <table border="1">
          <tr><th>Nome</th><th>Idade</th><th>Cidade</th></tr>
          <xsl:apply-templates />
        </table>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="pessoa">
    <tr>
      <td><xsl:value-of select="@nome"/></td>
      <td><xsl:value-of select="idade"/></td>
      <td><xsl:value-of select="cidade"/></td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```

Figura 2.23: Transformação XSLT

A primeira regra processa elementos chamados *pessoas*, que é o caso do elemento raiz. Esta regra gera a base do documento HTML, e despacha o processamento de regras para seus elementos filhos, através do elemento *apply-templates*.

A segunda regra processa elementos chamados *pessoa*, e para cada um deles ela gera uma linha de tabela. O documento HTML resultante é mostrado na figura 2.24[5].

```
<html>
  <body>
    <table border="1">
      <tr> <th>Nome</th><th>Idade</th><th>Cidade</th> </tr>
      <tr> <td>Carlos</td><td>25</td><td>Blumenau</td> </tr>
      <tr> <td>Juliana</td><td>19</td><td>Florianópolis</td> </tr>
    </table>
  </body>
</html>
```

Figura 2.24: Documento HTML Resultante

[5] A formatação do documento resultante foi alterada para melhor apresentá-lo

2.2.4 Servlets

Segundo a especificação JSR-154 (Coward e Yoshida, 2003), *servlets* são componentes de aplicações *web* baseadas na plataforma Java que permitem gerar conteúdo dinâmico. *Servlets* têm seu ciclo de vida gerenciado por contêineres.

O princípio de funcionamento dos *servlets* é baseado no paradigma de processamento de requisições. Um *servlet* fica à espera de uma requisição de um cliente, sendo ativado no momento de sua chegada. Ele a processa e retorna o conteúdo apropriado.

De acordo com Hunter e Crawford (2001), *servlets* devem implementar a interface *Servlet* (do pacote *javax.servlet*), que define, entre outros, o método *service*. O contêiner invoca este método sempre que uma requisição deve ser processada. Como parâmetro são passados dois objetos: o primeiro contém os parâmetros da requisição e o segundo permite montar a resposta a ser enviada. Para o tratamento específico do protocolo HTTP a API define a classe *HttpServlet*, cuja implementação do método *service* delega o processamento da requisição para métodos especializados para cada uma das requisições HTTP. Por exemplo, o método *doPost* é invocado quando do recebimento de requisições *POST*.

Um exemplo de *servlet* HTTP é apresentado na figura 2.25. Este *servlet* processa requisições *GET*, retornando conteúdo HTML.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ServletExemplo extends HttpServlet {
  public void doGet(HttpServletRequest req, HttpServletResponse res)
                                throws ServletException, IOException {
    res.setContentType("text/html");

    PrintWriter out = res.getWriter();
    out.println("<html>");
    out.println("<head><title>Teste de Servlets</title></head>");
    out.println("<body>Hello World</body>");
    out.println("</html>");
  }
}
```

Figura 2.25: Exemplo de um servlet

2.2.4.1 JSP

Escrever o conteúdo das respostas dentro da classe do *servlet* pode muitas vezes não ser o mais adequado. A tecnologia JSP (*JavaServer Pages*) (Roth e Pelegrí-Llopart, 2003) permite a escrita de páginas com conteúdo de linguagem de marcação com instruções de controle intercaladas. Uma página JSP pode conter qualquer conteúdo estático que se deseje, como por exemplo HTML, e também instruções Java ou *tags* especiais. Quando a página for acessada pela primeira vez ela é processada e um *servlet* é gerado a partir dela. A figura 2.26 apresenta um exemplo de página JSP, com instruções Java em destaque.

A figura 2.27 apresenta como ficaria o trecho de código no *servlet* gerado responsável pela geração do conteúdo da página da figura 2.26. Em destaque estão os trechos de código estático da página original.

```
<html>
  <head><title><%= request.getRemoteAddr() %></title></head>
  <body>
  <% for (int i=0; i<10; i++) { %>
    Teste
  <% } %>
  </body>
</html>
```

Figura 2.26: Exemplo de página JSP

```
PrintWriter out = res.getWriter();

out.println("<html>");
out.println("  <head><title>" + request.getRemoteAddr() + "</title></head>");
out.println("  <body>");
for (int i=0; i<10; i++) {
    out.println("    Teste");
}
out.println("  </body>");
out.println("</html>");
```

Figura 2.27: Exemplo de um servlet

2.2.5 Portais

Como apresentado na seção 1.1, portais são *sites* da *web* que simplificam e personalizam o acesso a conteúdo, aplicações e processos. As funcionalidades que um dividem-se, de acordo com Telang e Mukhopadhyay (2005), em três categorias :

- **Funcionalidades Pessoais**: Serviços que, em geral, necessitam de registro prévio e autenticação do usuário. Exemplos de serviços pessoais são: *e-mail*, salas de bate papo, fóruns de discussão e páginas personalizadas.
- **Funcionalidades de Informação**: Provêem informações de cunho geral a usuários do portal, como notícias, previsão do tempo e resultados de jogos. Em geral não necessitam de cadastro para acesso.
- **Funcionalidades de Busca**: A busca é um funcionalidade essencial para que a informação contida no portal seja encontrada. É necessário que a busca seja eficiente, ou seja, que a informação desejada apareça nos primeiros itens retornados pela busca.

Segundo com Hepper (2004), o número de portais de negócio vem crescendo e com isso surgiram várias arquiteturas para desenvolvimento de *portlets*, que são aplicações modularizadas dentro de portais. Posteriormente surgiu uma padronização através da especificação JSR-168 (Abdelnur e Hepper, 2003).

2.2.5.1 Portlets

Abdelnur e Hepper (2003) definem *portlets* como componentes que processam requisições e geram conteúdo dinâmico, na forma de fragmentos que são agregados para montar uma página de portal. *Portlets* são semelhantes a *servlets*, com a diferença que os últimos tem o intuito de criar aplicações *web* de propósito geral, e os primeiros possuem funcionalidades específicas para portais (Polgar et al., 2006).

Portlets são gerenciados por um contêiner, que cuida de seus ciclos de vida. Através do contêiner um servidor de portais interage com os *portlets* para obter os fragmentos de página que cada um provê, e então montá-los na página final, como mostrado na figura 2.28.

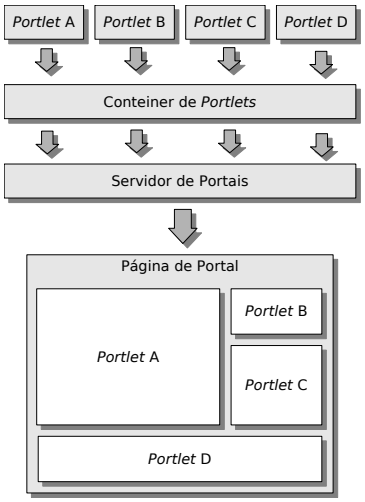

Figura 2.28: Agregação de *portlets* para criar uma página de portal

Polgar et al. (2006) diz que do ponto de vista de um usuário de portal, um *portlet* se parece como uma pequena aplicação dentro de uma página da *web*, que provê acesso a serviços ou conteúdo específicos. Cada *portlet* possui uma barra de título, com opções para minimizar e maximizar o portlet, da mesma forma que uma janela em um ambiente gráfico tradicional.

Portlets possuem modos de exibição bem definidos. Em geral três modos são oferecidos: um de visualização normal, um de edição, onde se configura aspectos do *portlet*, e um de ajuda. Outros modos podem ser suportados, dependendo do servidor de portais.

As requisições que um *portlet* recebe são de dois tipos:

- **Requisições de Renderização:** Este tipo de requisição acontece quanto o portal pede para o *portlet* seu conteúdo. Um fragmento de página é gerado, de acordo com o estado atual do *portlet*.
- **Requisições de Ação:** Este tipo de requisição acontece quando um usuário interage com o portlet. A partir dos dados do formulário preenchidos pelo usuário, que acompanham esta requisição, o estado do *portlet* pode ser alterado.

Um portal pede que um *portlet* lhe dê seu conteúdo a cada vez que a página é carregada. Por este motivo, não é permitido que requisições de renderização tenham efeitos colaterais no estado do *portlet*. Qualquer mudança de estado só pode acontecer a partir de uma requisição de ação.

A figura 2.29 mostra a interação entre o contêiner e seus *portlets* para atualizar a página do portal após uma interação de usuário. O usuário interage com o *Portlet A*, o que leva a uma requisição de renderização ser enviada para ele. Após isto, requisições de renderização são enviadas a todos os *portles* (inclusive o "A"), o portal monta uma nova página e a envia ao cliente.

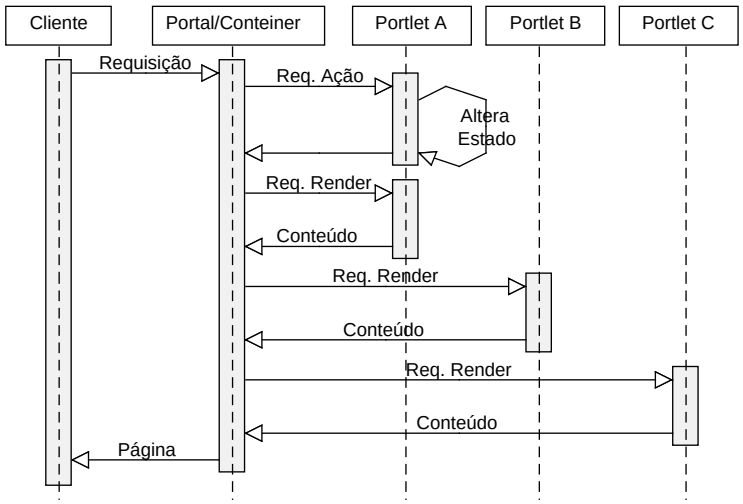

Figura 2.29: Interação entre contêiner e *portlets* após requisição de ação

Esta separação entre os tipos de requisição favorece o estilo de programação MVC. O *portlet* atua como um *controlador*, alterando o *modelo* a cada requisição de ação, e despachando a *visualização* a cada requisição de renderização. A *visualização* acessa o *modelo* para gerar o conteúdo dinâmico.

2.2.5.2 WSRP

WSRP (*Web Services for Remote Portlets*, ou Serviços *Web* para *Portlets* Remotos) é um padrão (Kropp et al., 2003) para portais acessarem e apresentarem *portlets* que estejam hospedados em servidores remotos.

Schaeckn (2002) discute a motivação para WSRP. O conteúdo de um portal é geralmente provido por serviços externos e apresentado por *portlets* hospedados no próprio portal. Este modelo dificulta a integração dinâmica de funcionalidades e fontes de informação em portais. Com WSRP, portais podem acessar *portlets* que estejam hospedados externamente, através de *proxies* genéricos, o que possibilita a integração dinâmica de quaisquer *portlets*. A diferença entre as duas abordagens é mostrada na figura 2.30.

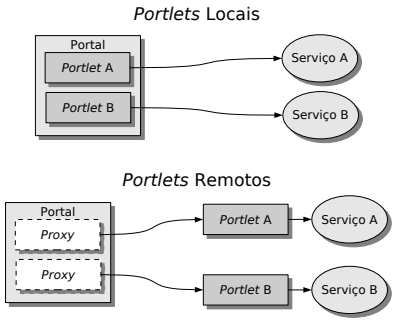

Figura 2.30: Diferença entre portais com *portlets* locais e remotos acessando serviços

A especificação caracteriza o portal como um *consumidor*, e o servidor que hospeda remotamente os *portlets* como um *produtor*. É definido então um mecanismo para que consumidores acessem um produtor via SOAP. Desta forma, os *proxies* não interagem diretamente com os *portlets* que referenciam, mas sim com o servidor WSRP onde os últimos estão hospedados. São definidas quatro interfaces que um servidor WSRP (produtor) pode implementar:

1. ***Service Description*:** Define um método para obter os metadados do produtor. Entre estes metadados está incluída a lista dos *portlets* oferecidos pelo produtor.
2. ***Markup*:** Define operações para obter conteúdo de um *portlet* do produtor e para processar interações do usuário.
3. ***Registration*:** Define operações para criar, atualizar e destruir cadastros de consumidores.
4. ***Portlet Management*:** Define operações para obter metadados de *portlets*, criar cópias e destruí-las.

Segundo Hepper et al. (2005), WSRP permite que servidores de portais compartilhem seus *portlets*, atuando como produtores, o que reduz custos administrativos e de manutenção. Isto significa que *portlets* JSR-168 hospedados em um portal podem ser acessados remotamente por outro portal.

2.2.6 Arcabouços MVC

MVC (Modelo-Visão-Controlador) é, segundo Buschmann et al. (1996), uma arquitetura que divide aplicações interativas em três componentes distintos: o *modelo*, que mantém o estado da aplicação, *visões*, que apresentam o modelo (ou partes dele) ao usuário, e *controladores*, que tratam as entradas do usuário para atualizar o modelo e apresentam os visualizadores apropriados para o usuário. A relação entre estes componentes é apresentada na figura 2.31.

Setas contínuas representem acessos diretos, enquanto que setas tracejadas indicam notificações. Assim, o controlador atua diretamente no modelo e na visão, podendo ser notificado de operações efetuadas pelo usuário (na visão). A visão acessa diretamente os dados do modelo e pode ser notificada de alterações neste.

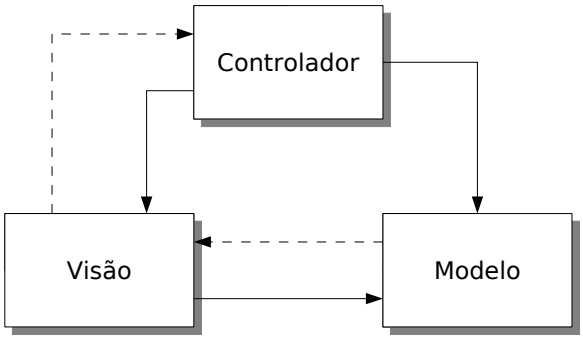

Figura 2.31: Arquitetura MVC

De acordo com Krasner e Pope (1988), o MVC surgiu no final da década de setenta, quando, após experiências de programação com o ambiente *Smalltalk-76*, viu-se que a modularização de aplicações daquela forma trazia grandes benefícios pois permitia isolar unidades funcionais umas das outras o máximo possível, tornando mais fácil o entendimento de cada unidade particular sem a necessidade do conhecimento das outras.

No âmbito das aplicações *web*, Thomas e Hansson (2005) citam que elas eram inicialmente feitas de forma monolítica, mas que com o tempo as idéias de MVC foram retomadas e adaptadas para este novo ambiente. Deste então, foram criados diversos arcabouços, ou seja, conjuntos de classes que incorporam um projeto abstrato para soluções para uma família de problemas relacionados (Johnson e Foote, 1988) para a construção de aplicações *web* seguindo o paradigma MVC. A seguir serão apresentados alguns dos mais usados atualmente.

2.2.6.1 Struts

Struts (Apache, 2000) é um projeto da Apache que consiste de um arcabouço de *software* para o desenvolvimento de aplicações *web* em Java, de acordo com o padrão de arquitetura de aplicações MVC.

Struts usa uma variação do MVC, específica para aplicações *web*, conhecida como *Model 2*. Nesta arquitetura, descrita por Husted et al. (2003), requisições são tratadas por um *servlet*, que instancia componentes (*Javabeans*) e redireciona a requisição para uma página JSP. Os componentes ficam disponíveis para a página e encapsulam o acesso a bases de dados. A figura 2.32 mostra o esquema desta arquitetura.

Struts fornece um *servlet* para atuar como controlador, e o fluxo da aplicação é especificado por descritores em XML. Uma biblioteca de *tags* é também oferecida para a escrita das páginas JSP, para simplificar o acesso aos componentes *Javabeans* e permitir validação dos dados.

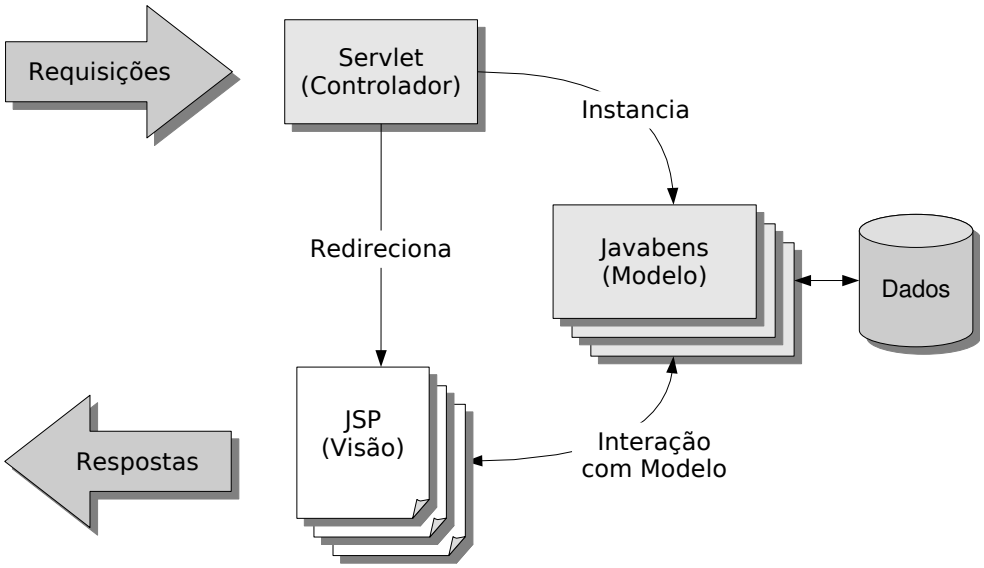

Figura 2.32: Arquitetura *Model 2*

2.2.6.2 JavaServer Faces

Segundo sua especificação (Burns e McClanahan, 2004), JavaServer Faces (JSF) é um arcabouço que foi projetado para facilitar significantemente o trabalho de escrever e manter aplicações que rodem em servidores Java e apresentam suas interfaces de usuário para clientes requisitantes. JSF provê facilidades das seguintes formas:

- Facilita a construção de interfaces de usuário a partir de um conjunto de componentes reutilizáveis.
- Simplifica a integração de dados da aplicação com a interface de usuário.
- Ajuda a gerenciar o estado da interface entre as requisições do servidor.
- Provê um modelo simples para tratar no lado servidor os eventos gerados no lodo cliente.
- Permite que novos componentes sejam facilmente feitos e reutilizados.

Os elementos da arquitetura JSF, descritos por Mann (2004), são apresentados a seguir.

- **Componentes de interface de usuário:** Objetos, mantidos no servidor, que provêm funcionalidades específicas para interagir com um usuário final. Eles possuem propriedades, métodos e eventos, e são organizados em uma árvore, que é apresentada geralmente como uma página.
- **Renderizador:** Responsável por apresentar os componentes de interface de usuário e traduzir entradas de usuário em propriedades de componentes.
- **Validador:** Responsável por garantir que os valores fornecidos por usuários possam ser aceitos.

- ***Beans* de suporte:** Objetos especializados que coletam valores dos componentes de interface de usuário e implementam métodos para tratar eventos.
- **Conversor:** Converte um valor de componente em texto para apresentação.
- **Eventos e Tratadores:** Eventos são gerados por componentes de interface de usuário. Estes componentes permitem que tratadores se registrem para capturá-los.
- **Mensagens:** Informações que são mostradas para o usuário. Praticamente todos os elementos da arquitetura podem gerar mensagens para mostrar informações ou erros.
- **Navegação:** Controla o fluxo de navegação dentro da aplicação.

A relação entre estes elementos é apresentada na figura 2.33

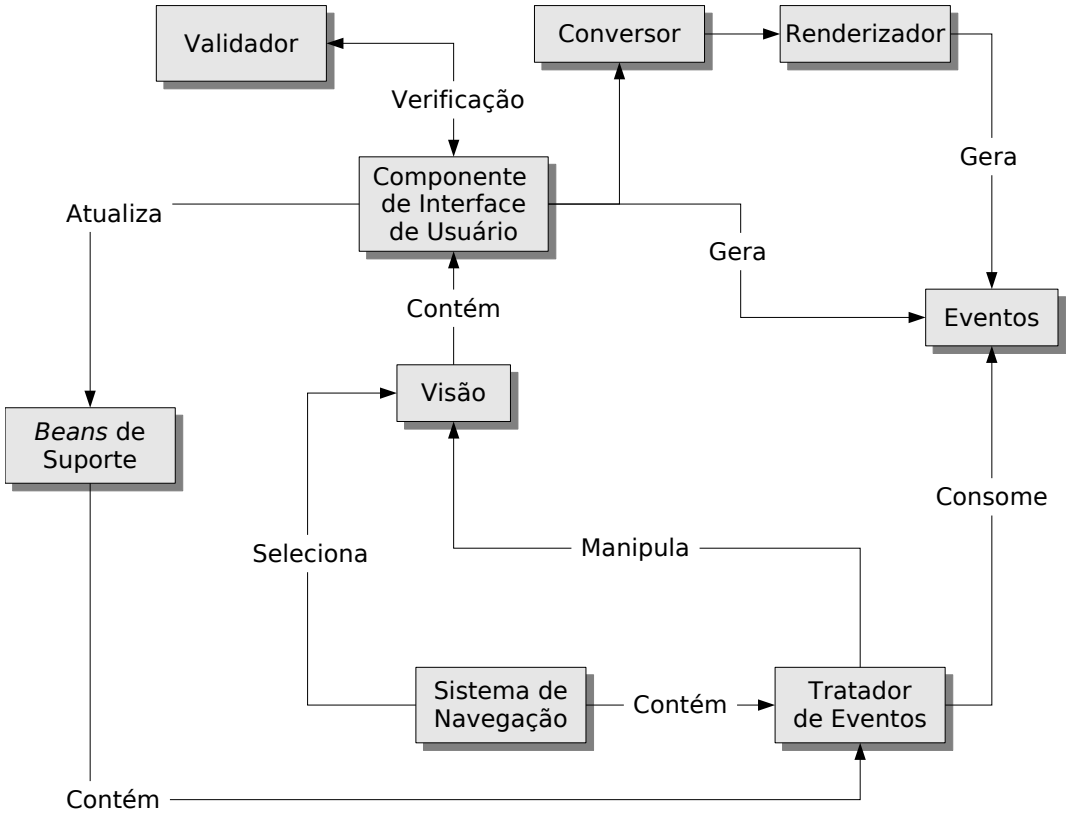

Figura 2.33: Arquitetura JSF

2.2.6.3 Rails

Rails (Rails, 2004) é um arcabouço de código aberto para o desenvolvimento de aplicações *web* que segue aproximadamente a arquitetura MVC. Ele faz uso da linguagem de programação Ruby, e foca na simplicidade, permitindo o desenvolvimento de aplicações robustas com menos programação e arquivos de configuração.

Em Thomas e Hansson (2005) é demonstrado o funcionamento de uma aplicação Rails. Um componente roteador trata a requisição e a envia para o controlador adequado. Controladores são instâncias da classe *ActionControler*. O controlador interage com o modelo, constituído por instâncias da classe *ActiveRecord*. Essas classes mapeiam tabelas do banco de dados. Rails permite um mapeamento automático, desde que os nomes das tabelas e colunas sigam um padrão convencionado. A flexibilidade da linguagem Ruby permite que este mapeamento seja dinâmico. Após a interação com o modelo, a visão apropriada é processada. Os componentes de visão são construídos como páginas HTML com seções de código Ruby inseridas. Este processo é chamado de ERb (*Embedded Ruby*, Ruby Embutido). Os componentes de visão acessam os de modelo e geram a página que é enviada na resposta. A figura 2.34 mostra processo.

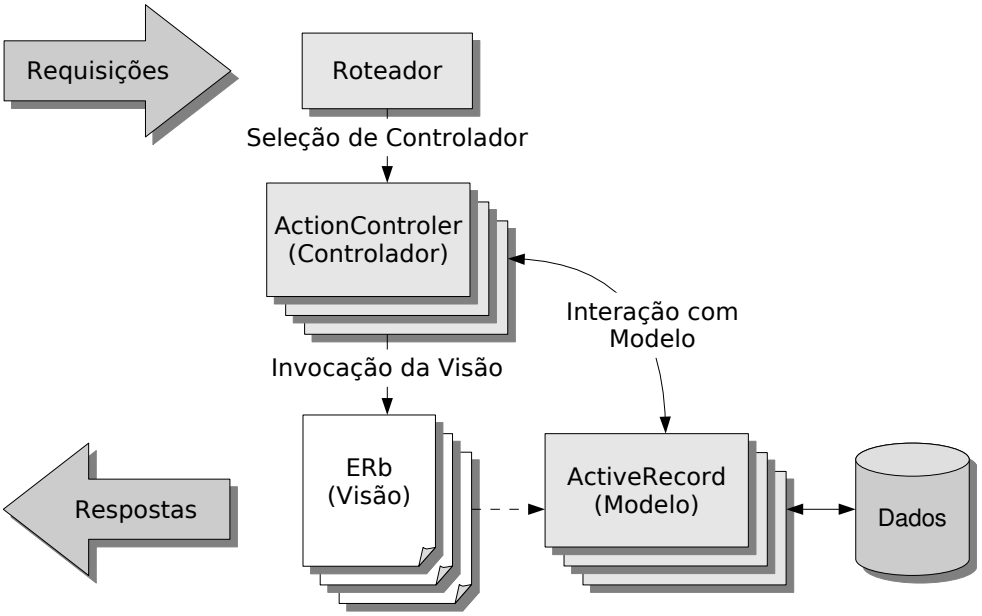

Figura 2.34: Arquitetura de uma aplicação Rails

2.2.6.4 Avaliação dos Arcabouços Estudados

Struts foi um dos primeiros arcabouços *web* a surgirem, e o JSF apareceu em seguida, como uma forma de se ter algo semelhante ao *Struts* mas padronizado, permitindo que qualquer desenvolvedor em qualquer contêiner padrão pudesse utilizá-lo. Atualmente o *Struts* está mudando de foco, indo mais na direção de um conjunto de componentes reutilizáveis para uso em JSF. Uma das críticas ao *Struts* e ao JSF é o grande número de arquivos de configuração em *XML* que se precisa criar. *Rails* é um arcabouço que consegue criar aplicações com praticamente nenhum arquivo de configuração, por ser baseado na linguagem *Ruby*. Características desta linguagem permitem que classes e métodos sejam criados em tempo de execução. A linguagem também facilita a meta programação. O primeiro item facilita a integração com banco de dados, enquanto que o segundo permite o abandono de arquivos de configuração. Entretanto *Rails* ainda sofre com problemas de desempenho e de falta de maturidade.

2.2.7 Conclusões sobre o Desenvolvimento de Aplicações Web

Como a proposta desta dissertação é a facilitação da integração de serviços *web* em portais, é natural que o método se baseie em *portlet* para isto, pois *portlets* são componentes padronizados específicos para isto. Neste caso, há poucos arcabouços com suporte direto para *portlets*, mas mesmo assim é possível empregar o paradigma MVC no desenvolvimento. Isto é facilitado por tecnologias como XSLT. Sendo tanto as mensagens como os descritores de serviços *web* documentos XML, eles podem ser transformados para serem apresentados a usuários de forma simples. Detalhes de como isto foi empregado na abordagem proposta são apresentados nos próximos dois capítulos.

Capítulo 3

Modelo Conceitual

Seguindo o objetivo proposto na seção 1.3.1, este capítulo detalha o modelo conceitual de uma abordagem para a integração de serviços *web* em portais. Em um primeiro momento a arquitetura é descrita de forma independente de qualquer tecnologia, desde que certos requisitos, apresentados no início da seção 3.1, sejam atendidos. Na segunda parte, esta arquitetura será adaptada para as tecnologias de serviços *web* e de componentes de portal atuais [1], mas sem detalhar aspectos de implementação, que serão apresentados no próximo capítulo. Deste modo, enquanto o modelo específico tem como requisito o uso das tecnologias para serviços *web* e portais, o modelo geral é mais abrangente, podendo ser adaptado a outras tecnologias existentes, como CORBA (OMG, 2004), ou novas tecnologias que venham a aparecer.

Esta abordagem propõe os seguintes passos para a disponibilização de um serviço *web* a usuários de um portal:

- Um componente de portal genérico deve ser instalado e configurado no portal. Esta etapa só precisa ser feita uma vez. Após sua implantação, o componente pode ser instanciado diversas vezes, para cada serviço *web* que se deseja invocar.
- Para cada serviço *web* que se deseja invocar deve-se fornecer o descritor do serviço desejado e escolher uma de suas operações (em casos de serviços que ofereçam mais de uma).
- Opcionalmente pode-se criar uma folha de estilo[2]para a apresentação da resposta ao usuário, uma vez que respostas de serviços *web* contêm apenas dados, sem nenhuma informação de como estes dados devem ser apresentados a usuários. Caso não seja criada a folha de estilo, os dados são mostrados de maneira crua, o que pode não ser adequado.

Desta forma, a necessidade de programação é eliminada, diminuindo assim o esforço necessário para se integrar um serviço a um portal. A figura 3.1 compara estes passos do modelo proposto com os tradicionais, apresentados na seção 1.4.

[1] SOAP, WSDL, UDDI e *Portlets*

[2] Folhas de estilo permitem adicionar estrutura gráfica a documentos que contém apenas dados.

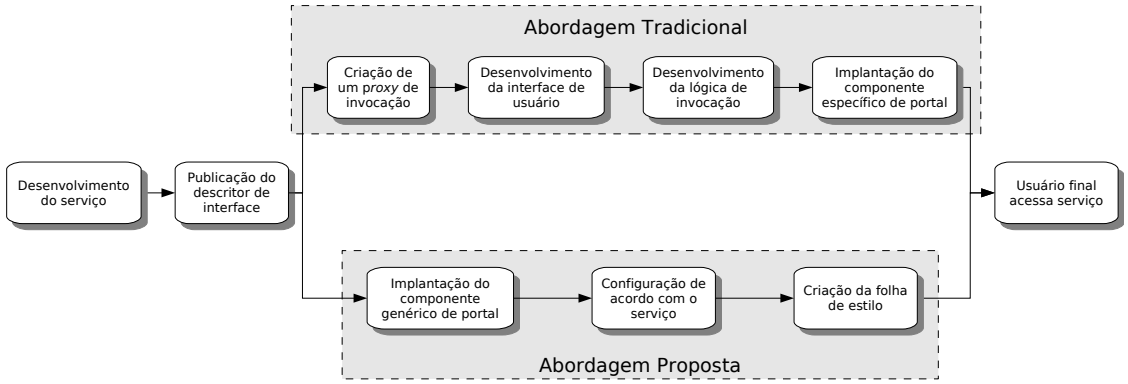

Figura 3.1: Diferenças entre a abordagem tradicional e a proposta

3.1 Arquitetura Independente de Tecnologia

Esta seção apresenta a arquitetura do modelo da abordagem proposta, sem se atrelar a nenhuma tecnologia, para serviços ou portais. O objetivo desta descrição é apresentar os componentes de forma mais abstrata, para que na próxima seção sejam incluídas as tecnologias no modelo. Este modelo depende dos seguintes requisitos:

1. Existem repositórios que permitem encontrar serviços através de buscas automatizadas. Estes repositórios devem ser acessíveis por uma API padronizada, de forma que diferentes repositórios possam ser pesquisados da mesma forma. Através destes repositórios deve ser possível descobrir o ponto de acesso de um serviço e a descrição de sua interface.

2. A interface dos serviços deve ser descrita por uma linguagem formal, passível de processamento por computador. A descrição deve ser completa o suficiente para que com apenas ela todos os parâmetros de um serviço sejam conhecidos, incluindo seus tipos de dados e estrutura.

3. Deve ser possível construir programaticamente invocações para serviços, ou seja, deve ser possível construir chamadas a serviços agregando-se dinamicamente os parâmetros necessários em alguma forma de requisição, para posterior envio ao ponto de acesso do serviço.

4. Portais devem ser modularizáveis, no sentido de que suas funcionalidades sejam providas por componentes que agreguem suas interfaces de usuário e lógicas internas. A interface entre os componentes e os portais deve ser padronizada, de forma que um componente possa ser instalado em diferentes portais, desde que siga o padrão. A interface de usuário é construída usando linguagens de marcação, que suportam campos de formulário para a entrada de dados por usuários. Cada componente de portal contribui com uma parte do que será a interface de usuário do portal como um todo.

Além disto, para a invocação de serviços esta abordagem suporta apenas aqueles serviços cujos parâmetros de entrada podem ser inseridos diretamente por pessoas. Serviços que esperem dados codificados de forma binária não são o alvo desta abordagem.

A figura 3.2 contém os elementos da arquitetura conceitual proposta para alcançar os objetivos propostos. O fluxo de troca de informações entre os módulos é indicado pela direção das setas. As próximas subseções detalharão cada um dos módulos.

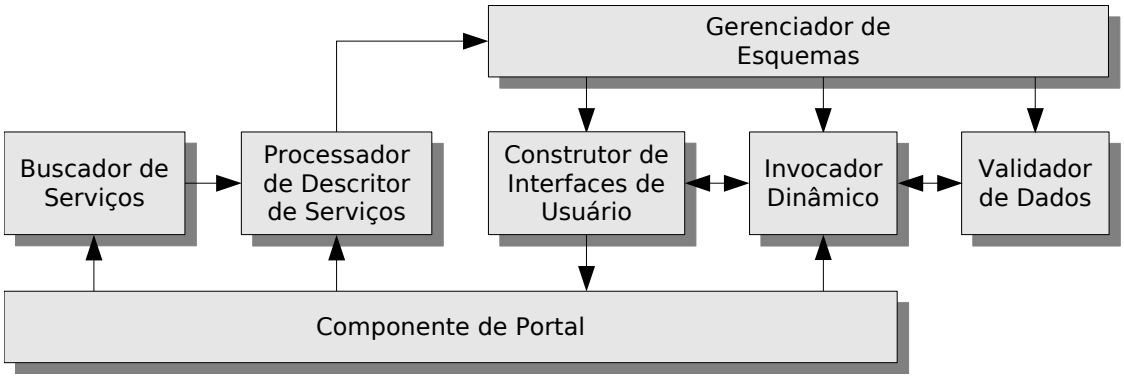

Figura 3.2: Arquitetura Independente de Tecnologia

A figura 3.3 ilustra o funcionamento da abordagem proposta. Inicialmente o portal não possui o componente genérico de invocação de serviços implantado. Em seguida, o componente é implantado, e no próximo passo o endereço de um descritor para um serviço *web* é inserido nele. Feito isto, componentes de entrada de dados são automática e dinamicamente gerados a partir deste descritor. Deste ponto em diante pode-se invocar o serviço, preenchendo os campos e clicando no botão enviar. Isto faz com que uma requisição seja dinamicamente criada e enviada ao serviço. Quando a resposta é recebida ela é apresentada ao usuário.

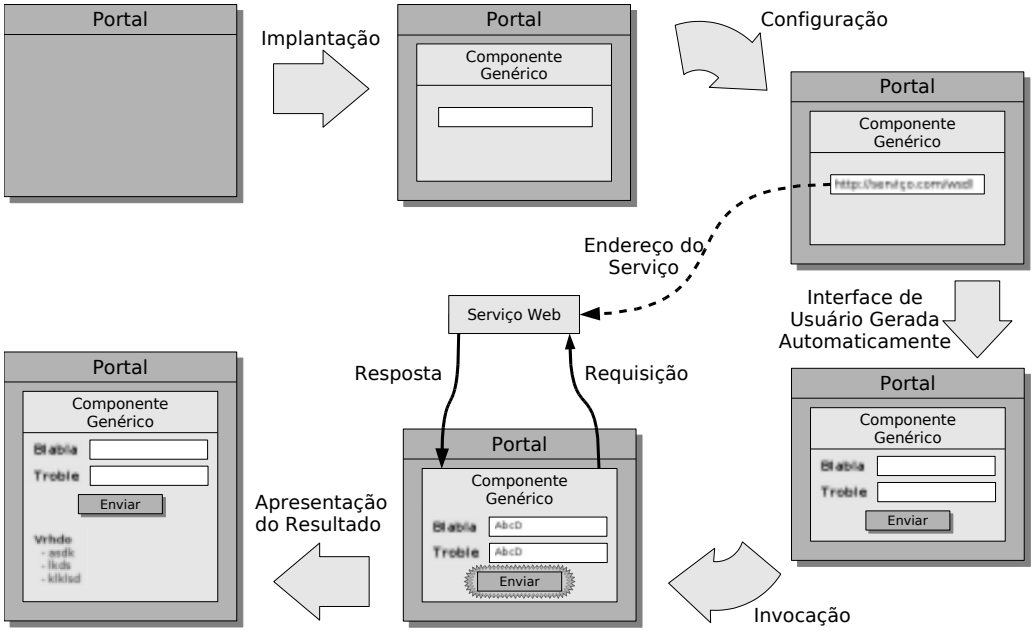

Figura 3.3: Passos da abordagem proposta

3.1.1 Componente de Portal

Todo o sistema é construído sobre um *Componente de Portal*. Isto é devido ao objetivo da integração com portais. O controle de todos os demais elementos é gerenciado por este elemento.

A interface de usuário de todo o sistema também é provida por este componente. Ela é dividida em duas partes:

- Uma parte fixa, de configuração, por onde um usuário efetuará buscas em repositórios por descritores de serviços, ou os fornecerá diretamente para processamento.
- Um parte gerada dinamicamente, com formulários de entrada de dados gerados de acordo com a descrição da interface de um serviço, e com resultados de invocações.

3.1.2 Buscador de Serviços

Usuários podem indicar diretamente uma descrição de serviço, cuja interface de usuário para a invocação será gerada dinamicamente, ou podem fazer uso de um mecanismo de busca em repositórios de serviços.

Serviços cadastrados em repositórios possuem uma descrição textual associada. Buscas são feitas com base nestas descrições, e o resultado é apresentado para que usuários escolham o serviço desejado. Após a escolha de um serviço ter sido feita, seu descritor é obtido e processado.

3.1.3 Processador de Descritor de Serviços

Este elemento processa descritores de serviço e extrai dados relevantes para sua invocação. São obtidos:

- Nome do serviço.
- Nomes das operações providas pelo serviço[3].
- Esquema de dados dos parâmetros das operações.

3.1.4 Gerenciador de Esquemas

Este módulo gerencia as informações obtidas anteriormente. Estas informações serão relevantes para a geração dinâmica do formulário de invocação do serviço, e para a posterior invocação deste.

[3]Em casos que mais de uma operação estiver disponível, uma escolha deve ser feita pelo usuário do sistema.

3.1.5 Construtor de Interfaces de Usuário

Este módulo gera dinamicamente interfaces de usuário. Estas interfaces são divididas em duas partes:

- **Entrada de Dados:** É composta por componentes de formulário gerados de acordo com o esquema de dados do serviço.
- **Apresentação de Resultados** Mostra o resultado das invocações dos serviços.

A figura 3.4 mostra como seria uma interface de usuário gerada por este componente, com a parte de entrada de dados acima, e o resultado das invocações apresentado abaixo.

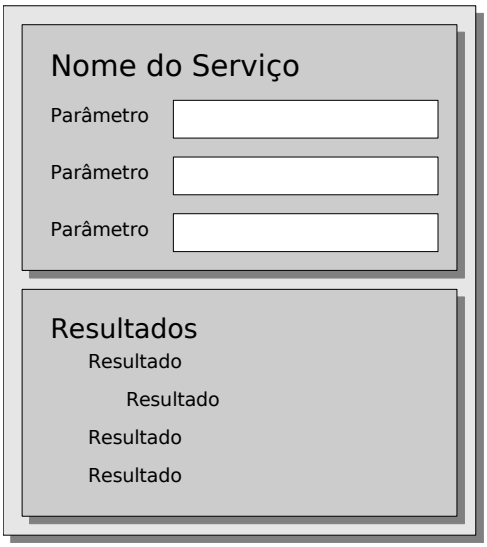

Figura 3.4: Simulação de interface de usuário gerada

3.1.6 Invocador Dinâmico

Com os dados fornecidos pelo usuário através da interface gerada dinamicamente, e o esquema de dados obtido do descritor do serviço, este módulo invoca dinamicamente o serviço. O resultado da invocação é enviado novamente ao construtor de interfaces de usuário, para ser apresentado ao usuário. Antes de se efetuar a invocação os dados são enviados ao *Validador de Dados*.

3.1.7 Validador de Dados

Este elemento recebe os dados inseridos pelo usuário e, com base no esquema de dados obtido do descritor do serviço, verifica se estes são válidos de acordo com os tipos esperados. No caso de haver discordâncias, o *Invocador Dinâmico* é avisado e aborta a invocação da operação. O usuário é então informado sobre os parâmetros inválidos.

3.2 Arquitetura para Serviços Web

A figura 3.5 mostra o modelo conceitual, que havia sido apresentado na figura 3.2, com as tecnologias de serviços *web* e portais indicadas.

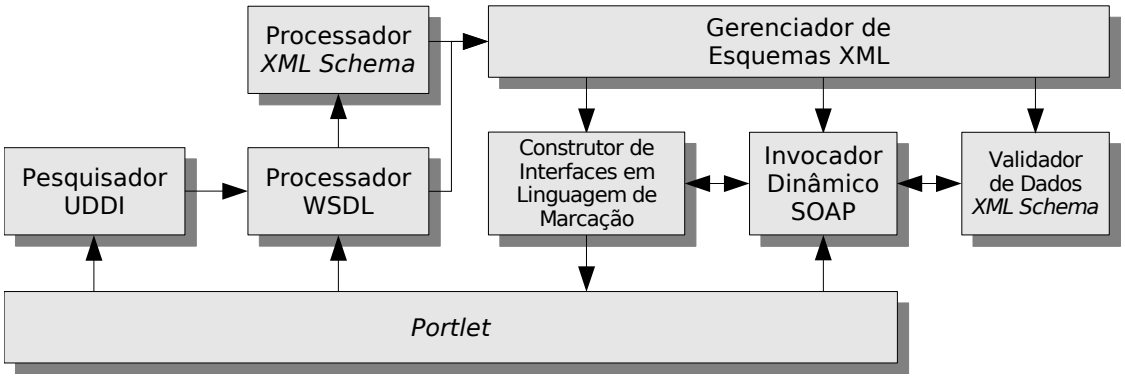

Figura 3.5: Arquitetura do sistema com indicação das tecnologias utilizadas por cada módulo

Este modelo atrelado a tecnologias permite uma melhor descrição das técnicas e algoritmos implementados. Cada um dos módulos do modelo abstrato possui um correspondente no modelo com tecnologias, exceto o *Processador de Descritor de Serviços*, que deu origem a dois módulos (*Processador WSDL* e *Processador XML Schema*).

3.2.1 Portlet

O *portlet* controla os demais módulos da arquitetura, e apresenta a interface de usuário do sistema. Um usuário pode utilizar o *portlet* em dois modos:

- **Configuração:** Para fazer buscas em repositórios UDDI, indicar diretamente o URL de um descritor de serviço, e escolher qual operação invocar no caso de serviços com mais de uma.
- **Invocação:** Para invocar serviços e visualizar o resultado.

Pelo modo de configuração usuários indicam o serviço *web* que querem utilizar no *portlet*. Esta indicação pode ser feita fornecendo-se diretamente um arquivo contendo a descrição WSDL do serviço, ou pela busca de serviços em repositórios UDDI, de onde é obtido o descritor WSDL. Ainda no modo de configuração é escolhida para qual operação do serviço será gerada a interface de usuário, no caso de o serviço oferecer mais de uma operação.

O modo de invocação mostra os componentes de interface de usuário, gerados automaticamente, para o serviço escolhido. Os dados inseridos nestes componentes são enviados para o invocador dinâmico, que devolve o resultado da invocação do serviço, apresentada para o usuário em seguida.

3.2.2 Pesquisador UDDI

Nesta abordagem, a busca por serviços é feita em repositórios UDDI, dada a sua grande utilização. Como mencionado na seção 2.1.6.2, referências para descritores de serviços *web* são armazenadas como elementos *tModel*. Elementos *tModel* que contenham descrições de serviços devem possuir a classificação *uddi-org:types*, com o valor *wsdlSpec* (Curbera et al., 2002). Desta forma a busca por serviços *web* em repositórios deve limitar-se àqueles serviços que contenham elementos *tModel* desta forma.

A API de acesso a repositórios UDDI permite executar buscas com esta forma de filtragem. Assim, após um usuário escrever as palavras-chave de sua busca, os resultados são apresentados e ele pode escolher um dos serviços resultantes.

3.2.3 Processador WSDL

O uso de WSDL como linguagem de descrição é natural, dado que ela é a linguagem especificada para isto em serviços *web*.

Durante o processamento de um documento WSDL, parte dele (a descrição dos tipos de dados das mensagens, feita em *Schema*) é passada para um processador dedicado. A principal parte do processamento do processador de WSDL é a identificação dos serviços e das suas das operações.

Como mencionado na seção 2.1.5, um documento WSDL pode conter vários serviços. Um serviço é constituído por um ou mais *ports*. Um *port* é uma ligação disponível em um endereço. Uma ligação é um vínculo de um tipo de *port* abstrato com um protocolo. Um tipo de *port* pode conter várias operações, cada uma delas com mensagens de entrada e de saída. Por fim, cada mensagem é composta por uma ou mais partes, que referenciam tipos declarados em *XML Schema*. Esta estrutura pode ser vista no diagrama da figura 3.6.

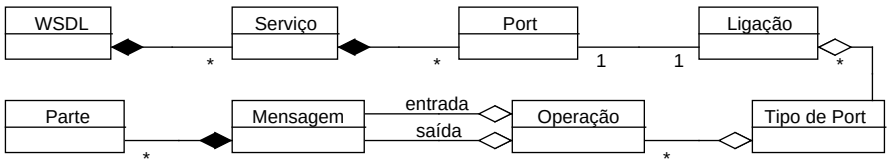

Figura 3.6: Diagrama UML dos elementos de um documento WSDL

Durante o processamento de um documento WSDL, um grafo simplificado é gerado. A estrutura deste grafo foi criada especificamente para suportar as necessidades do modelo proposto. O processamento pode então ser visto como uma simplificação do grafo hipotético que representa um documento WSDL, eliminando-se informações desnecessárias. Como só são tratados serviços SOAP sobre HTTP, todos os *ports* desnecessários (e suas ligações) são removidos. As operações dos tipos de *port* podem ser associadas diretamente aos vértices que representam serviços. O agrupamento dos

parâmetros em mensagens de entrada e saída também pode ser eliminada. Como resultado tem-se um *grafo dirigido*, contendo os serviços, suas operações e seus parâmetros, como mostrado na figura 3.7.

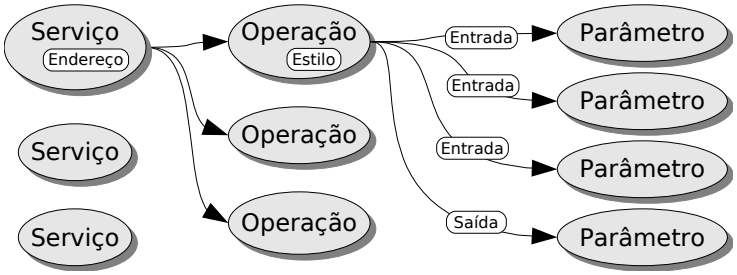

Figura 3.7: Estrutura simplificada dos serviços e operações de um documento WSDL

Neste grafo constam apenas serviços SOAP sobre HTTP. Cada serviço é rotulado por seu endereço de invocação e cada operação por seu estilo[4]. Há três tipos de vértices:

- **Vértice de Serviço:** Representa um serviço.
- **Vértice de Operação:** Representa uma operação de um serviço.
- **Vértice de Parâmetro:** Representa um parâmetro de uma operação.

Cada parâmetro é referenciado por apenas uma operação, assim como cada operação é referenciada por apenas um serviço. As arestas que ligam operações a parâmetros são rotuladas para indicar se o parâmetro é de entrada ou saída. O processo de geração deste grafo, a partir de um documento WSDL, é demonstrado no algoritmo 3.1.

Algoritmo 3.1 Geração de grafo de serviços

```
1:  para cada serviço s que possua ports SOAP faça
2:      criar um vértice do tipo serviço, rotulado com o nome de s
3:      rotular o vértice com o endereço de acesso do port SOAP de s
4:      para cada operação o da ligação do port de s faça
5:          criar um vértice do tipo operação, rotulado com o nome e o estilo de o
6:          criar uma aresta ligando o vértice de s a ao vértice de o.
7:          para cada parte p declarada na mensagem de entrada de o faça
8:              criar um vértice do tipo parâmetro, rotulado com o nome de p e o nome de seu tipo.
9:              criar uma aresta ligando o vértice de o ao vértice de p, rotulada como entrada.
10:         fim para
11:         para cada parte p declarada na mensagem de saída de o faça
12:             criar um vértice do tipo parâmetro, rotulado com o nome de p e o nome de seu tipo.
13:             criar uma aresta ligando o vértice de o ao vértice de p, rotulada como saída.
14:         fim para
15:     fim para
16: fim para
```

[4]Que influencia na forma de se montar a invocação, como apresentado na seção 2.1.5.5

3.2.4 Processador XML Schema

A complexidade de documentos *XML Schema* demanda um processador exclusivo. O trabalho deste processador é gerar um *grafo dirigido*, cuja estrutura foi desenvolvida para este trabalho, que represente simplificadamente a estrutura do esquema. Neste grafo há dois tipos de vértices:

- **Vértice de Elemento:** Representa um elemento ou atributo XML.
- **Vértice de Tipo:** Representa um tipo de dado.

Cada vértice de elemento é ligado a um vértice de tipo, indicando o tipo do elemento. Vértices de tipo podem ou não se ligar a vértices de elementos. Caso o tipo representado seja composto, então cada um dos vértices de elementos filhos será adjacente ao vértice de tipo composto. Em caso contrário, o grau de emissão do vértice de tipo será zero, mas o vértice será marcado com informações sobre as restrições do tipo, como os valores que o tipo aceita, tamanho da entrada, etc. A figura 3.8 mostra o grafo gerado a partir do *XML Schema* da figura 2.5 (página 14). O elemento *idade* possui uma marcação por representar um atributo.

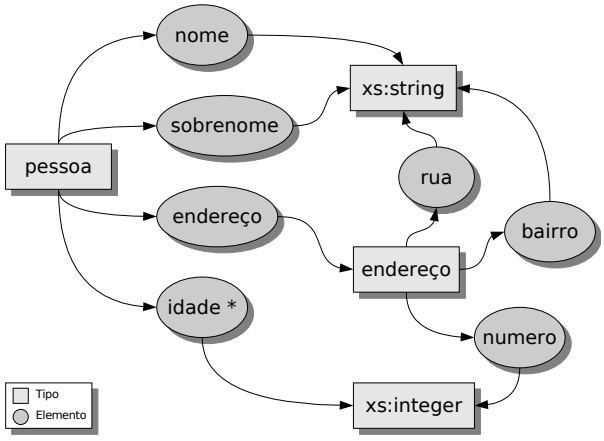

Figura 3.8: Grafo gerado a partir de documento *XML Schema* da figura 2.5 (página 14)

O algoritmo 3.2 mostra como é gerado o grafo de tipos a partir de um documento *XML Schema*.

3.2.5 Gerenciador de Esquemas XML

O gerenciador de esquemas liga o grafo de serviços, gerado pelo processador WSDL, com o grafo de tipos, gerado pelo processador *XML Schema*. A estrutura resultante é o *grafo de esquema completo*, que os módulos de construção de interface e de invocação de serviços utilizam. Este processo acontece de acordo com o algoritmo 3.3.

A figura 3.9 mostra um exemplo do resultado final.

[5] Como os do grafo de tipos.

Algoritmo 3.2 Geração de grafo de tipos

```
1: para cada elemento e declarado no esquema faça
2:    criar um vértice de elemento, rotulado com o nome de e
3: fim para
4: para cada atributo a declarado no esquema faça
5:    criar um vértice de elemento, rotulado com o nome de a.
6:    marcar o vértice como "atributo"
7: fim para
8: para cada tipo t declarado no esquema faça
9:    criar um vértice de tipo
10:   se t possuir nome então
11:      rotular o vértice com o nome de t
12:   senão
13:      rotular o vértice com o nome do elemento que envolve t
14:   fim se
15:   se t for um tipo simples então
16:      rotular o vértice com as restrições declaradas em t
17:   senão
18:      para cada elemento e pertencente a t faça
19:         criar uma aresta ligando o vértice de t ao vértice de e
20:         se as cardinalidades mínima e máxima de e em t forem diferentes de 1 então
21:            rotular a aresta com as cardinalidades
22:         fim se
23:      fim para
24:   fim se
25: fim para
26: para cada vértice de elemento v gerado faça
27:    criar uma aresta ligando v ao vértice de tipo correspondente
28: fim para
```

Algoritmo 3.3 Geração do grafo de esquema completo

```
1: para cada vértice de parâmetro p do grafo de serviço faça
2:    transformar p em um vértice de elemento[5]
3:    criar uma aresta ligando p ao vértice de tipo do grafo de tipos correspondente
4: fim para
5: eliminar os vértices (e arestas correspondentes) do grafo de tipos que forem inacessíveis
```

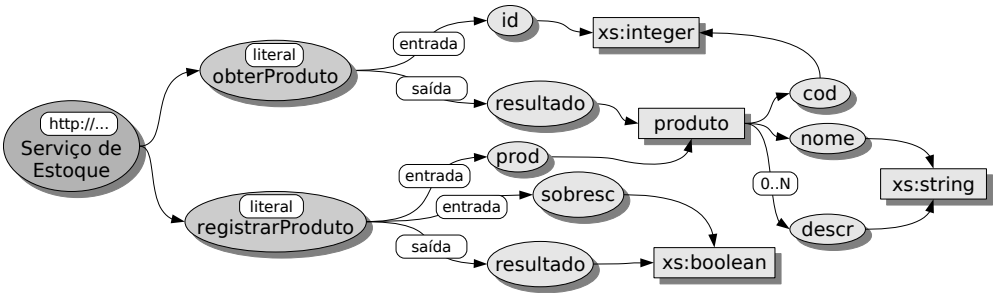

Figura 3.9: Exemplo de grafo de esquema completo

Neste exemplo é mostrado um serviço com duas operações: *obterProduto* e *registrarProduto*. A primeira possui um parâmetro de entrada: *id* do tipo *xs:integer*, e um de saída: *resultado*, do tipo *produto*. A segunda operação possui dois parâmetros de entrada: *prod*, do tipo *produto* e *sobresc*, do tipo *xs:boolean*, e um de saída: *resultado*, também do tipo *xs:boolean*. O tipo *produto* é composto por três elementos: *cod*, do tipo *xs:integer*, e *nome* e *descr*, ambos do tipo *xs:string*, mas *desc* pode ser repetido de zero a ilimitadas vezes.

3.2.6 Construtor de Interfaces em Linguagem de Marcação

Este módulo gera a interface de usuário para uma dada operação de um serviço, em linguagem de marcação (por exemplo: HTML, XHTML, WML), que codifica a interface de usuário, elementos que representam componentes de entrada de dados. Através desta interface usuários podem invocar o serviço preenchendo os parâmetros necessários. Além disto, este módulo também gera uma representação em linguagem de marcação do resultado da invocação do serviço.

3.2.6.1 Geração da interface de invocação

A interface é gerada na forma de um documento em linguagem de marcação, gerado pelo algoritmo 3.4, a partir do grafo de esquema completo, produzido pelo gerenciador de esquemas. Caso o grafo de esquema contenha mais de uma operação, o usuário deverá escolher uma delas, e apenas o subgrafo correspondente a esta operação será considerado.

Seguindo este algoritmo, a interface de usuário gerada a partir do grafo da figura 3.9, para a operação *registrarProduto*, seria semelhante à figura 3.10. O campo de entrada de dados para *cod* aceita somente números inteiros, *descr* possui a opção de se criar dinamicamente mais campos, e *sobresc* só aceita os valores booleanos *true* ou *false*.

Os dados inseridos pelo usuário são validados de acordo com o esquema, e ficam disponibilizados como uma árvore. Dados ficam em folhas, rotuladas com o nome do elemento correspondente, e elementos complexos em nós, como no exemplo da figura 3.11, que mostra a árvore gerada pela interface mostrada na figura 3.10.

[6]Em caso de chamadas recursivas, são concatenados os nomes de todos os vértices de elemento no caminho.

Algoritmo 3.4 Construção de interface de usuário

1: seja *o* um vértice de *operação*.
2: **para cada** vértice (de *elemento*) *e* adjacente a *o* **faça**
3: criar um elemento de texto estático, contendo o nome rotulado em *e*
4: seja *t* o vértice (de *tipo*) adjacente a *e*
5: **se** *t* não possuir vértices adjacentes **então**
6: criar um campo de entrada de dados, de acordo com o tipo de *t*
7: usar o nome de *o* como identificador do campo de entrada de dados[6]
8: aplicar informações de restrição do tipo de *t* ao campo de entrada de dados
9: **senão**
10: executar este algoritmo recursivamente, tomando *t* como *o*
11: **fim se**
12: **se** a aresta de *o* a *e* possuir informações de cardinalidade **então**
13: criar um botão para a criação de novas cópias dos componentes gerados para *e*
14: **fim se**
15: **fim para**
16: **se** estiver no meio de uma invocação recursiva **então**
17: **retornar**
18: **fim se**
19: criar um botão para invocar o serviço

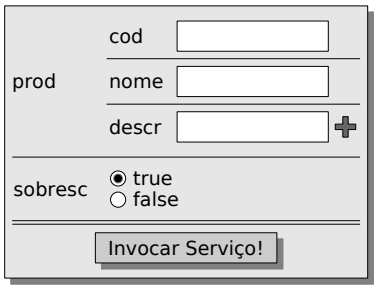

Figura 3.10: Interface gerada para o grafo da figura 3.9

3.2.6.2 Geração da interface para apresentar o resultado da invocação

Outro trabalho deste módulo é mostrar o resultado de invocações SOAP ao usuário. Isto é feito pela conversão do envelope SOAP que contém a resposta em um documento de linguagem de marcação. O processo é feito trocando-se todos os elementos XML do envelope (a partir do elemento *body*) por elementos de marcação que mantenham a hierarquia dos dados na hora da apresentação.

Uma folha de estilo pode ser fornecida para indicar como os elementos XML são transformados em elementos de marcação. Assim pode-se apresentar os resultados de forma mais adequada em relação ao seu conteúdo, por exemplo como uma tabela.

3.2.7 Invocador dinâmico SOAP

A partir da árvore dos dados inseridos pelo usuário nos campos de entrada de dados (mostrada na seção 3.2.6.1), e com a ajuda do grafo de esquema completo (gerado de acordo com a seção3.2.5), o

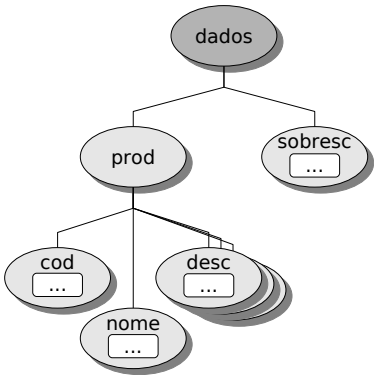

Figura 3.11: Árvore de dados da interface da figura 3.10

invocador dinâmico cria um envelope SOAP. Inicialmente o grafo de esquema é analisado para se obter o nome da operação a ser invocada. Em seguida, a árvore de dados é percorrida em profundidade, e seus nodos são convertidos em elementos do envelope SOAP. O algoritmo 3.5 descreve formalmente a execução deste processo.

Algoritmo 3.5 Criação dinâmica de envelopes SOAP

1: criar a estrutura básica de um envelope SOAP
2: seja *b* o elemento XML *body* deste envelope
3: seja *o* o vértice que representa a operação no grafo de esquema completo
4: seja *r* o nó raiz da árvore de dados
5: seja *x* um novo elemento XML criado com o mesmo nome de *o*
6: incluir *x* em *b*
7: **para cada** nó *n* filho de *r* **faça**
8: seja *e* um novo elemento XML criado com o mesmo nome de *n*.
9: adicionar *e* a *x*
10: **se** *n* não possuir nós filhos **então**
11: inserir o dado contido em *n* em *e*
12: **senão**
13: invocar o algoritmo recursivamente, a partir da linha 7, usando *n* como *r*
14: **fim se**
15: **fim para**
16: **se** estiver no meio de uma invocação recursiva **então**
17: **retornar**
18: **fim se**

Depois de gerado, o envelope é enviado por HTTP (ou qualquer que seja o mecanismo de transporte utilizado). Neste momento cabe ao servidor que implementa o serviço *web* executar a lógica a ele associada e retornar a resposta, na forma de um outro envelope SOAP. Enquanto isto o processo local fica esperando. Ao chegar a resposta, esta é enviada ao gerador de interfaces de usuário para apresentá-la, como mostrado na seção 3.2.6.2.

3.2.8 Validador de Dados XML Schema

Antes da criação do envelope SOAP, a árvore de dados inseridos pelo usuário é validada de acordo com o esquema. O *Validador de Dados XML Schema* percorre as folhas da árvore e verifica a concordância do dado inserido com o tipo esperado pelo esquema. Uma lista com as eventuais entradas discordantes é gerada e posteriormente apresenta ao usuário. Enquanto houver problemas a invocação do serviço não ocorre.

3.3 Trabalhos Relacionados

Neste capítulo serão apresentados trabalhos, incluindo tanto projetos acadêmicos e de pesquisa como projetos comerciais, que tratam problemas semelhantes ao tratado por este trabalho.

3.3.1 Projetos Acadêmicos e de Pesquisa

Esta seção apresenta trabalhos acadêmicos ou projetos que objetivam automatizar ou simplificar a disponibilização de serviços para usuários.

3.3.1.1 Abordagem Baseada em Templates para a Configuração em Massa de Aplicações de Comércio Eletrônico Orientadas a Serviços

Neste artigo, Zhu e Zheng (2005) apresentam um conjunto de ferramentas de desenvolvimento de *software* para a criação de aplicações de comércio eletrônico, suportadas pela SOA. Os autores predefinem um diretório de serviços, na forma de *templates*, que são necessários para a maioria dos sistemas de comércio eletrônico, como sistemas de finanças e identificação baseados na *Web*, vendas e propaganda, gerenciamento da relação com o cliente, gerenciamento de cadeia de produção e automação de escritório. A integração destes *templates* de serviços de aplicação permite criar uma solução coesa para tratar um dado negócio.

Cada um destes *template* abstrai as funcionalidades de um módulo ou subsistema, podendo ser vistos como componentes. Uma linguagem de definição própria foi desenvolvida, onde especificam-se informações como fluxos de execução, entrada e saída de dados e tabelas de banco de dados relacionadas. Desta forma aplicações genéricas podem ser construídas e ligadas a diferentes *templates* para proverem diferentes funcionalidades.

A abordagem proposta nesta dissertação diferencia-se da do artigo por não necessitar da criação prévia de *templates*. Assume-se que os próprios serviços já tenham uma granularidade suficientes para que sejam úteis por si só para a integração em portais, bem como que sua descrição seja suficiente para a geração dos componentes de entrada de dados para usuários.

3.3.1.2 Arquitetura Genérica de Software para Disponibilização de Aplicações Web para Dispositivos Móveis

Este trabalho, de Fiorini (2006), propõe uma arquitetura para o desenvolvimento de aplicações *web* que permite gerar o conteúdo visual das aplicações dinamicamente, de acordo com a linguagem que marcação suportada pelo cliente. Isto é feito através de *templates* XSLT, escolhidos dinamicamente de acordo com as linguagens de marcação que o navegador informa suportar. Desta forma, a interface de usuário é gerada dinamicamente, conforme as capacidades do dispositivo usado (assume-se que o navegador instalado no dispositivo informe corretamente suas capacidades).

A arquitetura permite, através de uma interface baseada em documentos XML, o acesso a sistemas legados, bastando para isso que se escreva os *templates* XSLT, para cada tipo de linguagem de marcação que se pretenda suportar, e para cada tipo de documento XML que se deseje apresentar ao usuário.

Em comparação, a abordagem proposta também permitiria o acesso a sistemas legados, desde que para eles seja desenvolvida uma interface de acesso via serviços *web*. O acesso a dispositivos móveis seria possível caso o portal oferecesse esta funcionalidade.

3.3.1.3 C-Cube Framework

Segundo Canfora et al. (2005), o C^3 (Criação, Certificação e Classificação de Serviços) é um arcabouço que permite:

- **Publicação de serviços:** Permite que serviços sejam registrados para posterior descoberta. Cada serviço deve possuir uma descrição textual, uma descrição funcional (com anotações semânticas, baseadas em ontologias) e uma descrição de QoS.
- **Descoberta Semântica de Serviços:** Para tanto, cada operação dos serviços é descrita em termos de tipos de informação de entrada e de saída. São levadas em conta as diferenças entre a categorias ontológicas de tipo denominadas *conceitual* e *de informação*. Um tipo conceitual representa um tipo do domínio de aplicação, enquanto um tipo de informação trata-se apenas de um conjunto de atributos. Cada tipo conceitual pode corresponder a diferentes tipos de informação, dependendo do contexto aplicado.
- **Negociação de SLA:** Permite garantir os acordos de QoS estabelecidos.
- **Composição levando em conta QoS:** Durante buscas de serviços, níveis de qualidade de serviço podem ser especificados. Este parâmetros também são levados em conta no momento de compor serviços.

Não há menção sobre a interface de usuário dos serviços providos pelo arcabouço. A abordagem proposta nesta dissertação poderia ser aplicada e, para melhor aproveitar os recursos oferecidos, ser estendida para levar em conta as anotações semânticas fornecidas.

3.3.1.4 Open Knowledge Initiative

A OKI (*Open Knowledge Initiative*, ou Iniciativa de Conhecimento Aberto) é uma organização responsável pela especificação de interfaces de *software* para SOAs, baseada em definições de alto nível (Open Knowledge Initiative, 2006). Estas interfaces são conhecidas como OSDIs (*Open Service Interface Definitions*, ou Definições Abertas de Interfaces de Serviço), e cada uma delas descreve um serviço computacional lógico. O uso de interfaces neutras separa a lógica de negócios da real implementação dos serviços, o que é a base de qualquer SOA. Já foram definidas interfaces para serviços de:

- **Repositório:** Define o armazenamento e obtenção de conteúdo digital.
- **Agentes:** Suporta a criação, busca e remoção de Agentes e Grupos.
- ***Scheduling*:** Provê meios para associar atividades específicas a Agentes.
- ***Workflow*:** Provê meios para definir processos compostos de uma série de passos.
- **Troca de Mensagens:** Provê meios para o envio e recebimento de mensagens.
- **Gerenciamento de Curso:** Suporta a criação de um catálogo de curso.
- **Avaliação:** Suporta a criação, organização, administração, avaliação armazenamento e obtenção de informações de avaliação.
- **Autenticação:** Suporta a invocação de um processo de autenticação.
- **Autorização:** Provê meios para definir quem é autorizado a fazer o quê e quando.
- **Arquivamento:** Provê armazenamento e gerenciamento de arquivos e diretório de forma independente de plataforma.

Caso estas especificações de interfaces de serviços ganhem aceitação, interfaces de usuário para muitos desses serviços tornar-se-ão importantes, e a abordagem proposta por este trabalho poderá ser empregada.

3.3.1.5 XML Forms Generator

Plug-in para o ambiente *Eclipse*, baseado em padrões e dirigido a dados, que gera formulários *XForms* dentro de documentos XHTML a partir de documentos WSDL ou XML *Schemas* (Kelly et al., 2006). Com este *plug-in* pode-se gerar formulários estaticamente a partir de um arquivo, ou seja, cada formulário gerado é um arquivo, que pode ser usado posteriormente para o devido fim.

Este *plug-in* gera código estaticamente, que deve ser posteriormente compilado em uma aplicação. A abordagem proposta atua de forma diferente. Não há geração de código e os componentes de interface de usuário são gerados e arranjados dinamicamente. Em geral abordagens estáticas são

mais eficientes, mas quando se trata de serviços *web* o ganho é pequeno. Tanto abordagens estáticas como dinâmicas para a invocação de serviços *web* precisam montar mensagens SOAP em Bray et al. (2004), o que é um processo bastante custoso, comparado ao tempo gasto com a geração dinâmica da invocação.

3.3.1.6 Xml Portlet

Conforme descrito no *site* do projeto (XML Portlet, 2006), este projeto almeja prover um *portlet* com um mecanismo simples para transformar dados XML em HTML para a apresentação para o usuário. Estes dados podem vir de diversas fontes, como provedores RSS, bancos de dados XML ou aplicativos de troca de mensagens. O *portlet* do projeto é compatível com o padrão JSR-168 (Abdelnur e Hepper, 2003). Os desenvolvedores também alegam que o uso deste *portlet* encoraja o padrão de arquitetura de aplicações *Modelo-Visão-Controlador* (MVC)[7], uma vez que há uma separação obrigatória entre os dados, a apresentação e o controle de fluxo. O *portlet* que o projeto provê deve ser estendido e a lógica de controle de fluxo deve ser programada.

Há algumas semelhanças entre o método utilizado neste projeto e a abordagem proposta por esta dissertação, como o fato de ambos permitirem a integração de informações externas em portais através de um *portlet*. Entretanto o *Xml Portlet* espera que a lógica seja programada, o que é devido ao fato de as fontes de dados integradas não possuírem por si só uma lógica associada. A abordagem proposta, que por sua vez integra serviços *web*, que já possuem inerentemente uma lógica associada. Desta forma não há a necessidade de mais programação.

3.3.1.7 Xydra

Xydra (Chipara e Slominski, 2003) é um *Servlet* que provê um invocador de serviços *web* baseado em XHTML. Ele toma um WSDL como entrada, gera um formulário XHTML para os usuários preencherem, converte as entradas em uma mensagem XML que é enviada ao serviço *web*, e mostra o resultado a seguir.

Este trabalho faz parte de um projeto maior, descrito por Gannon et al. (2005), cujo intuito é permitir que usuários finais possam encontrar e usar serviços *Grid*, sem a necessidade de um grande conhecimento sobre as tecnologias envolvidas.

Sua principal característica é a de possuir dois modelos de dados. O primeiro, chamado *TreePath*, trata os dados na forma de pares nome-valor, organizados de forma hierárquica. O segundo, chamado *OntoBrew*, é baseado na *engine Protégé*, e usa ontologias para descrever serviços, permitindo uma melhor validação dos dados.

Este projeto funciona de forma bem parecida com a abordagem proposta por esta dissertação, com a diferença de a geração dinâmica de interface de usuário estar alojada em um *servlet* em vez de em um *portlet*. Para que fosse integrado em portais, o *servlet* do Xydra precisaria adaptado para ser incluído dentro de um *portlet*.

[7] Apresentado na seção 2.2.6

3.3.2 Ferramentas Comerciais

Nesta seção são apresentadas ferramentas comerciais que trabalham de forma a resolver o problema de simplificar a disponibilização de serviços para usuários.

3.3.2.1 Generic SOAP Client

Trata-se de um cliente simples para acessar serviços *web* através de um navegador *web*. Ele foi desenvolvido para demonstrar que o uso de invocações dinâmicas de serviços *web* é viável, e para demonstrar as capacidades da biblioteca *SOAP Client Library* para a invocação de serviços (SQLData System, 2003). Entretanto, por se tratar apenas de uma aplicação-exemplo para uma biblioteca, é pouco flexível, principalmente se o intuito for a sua inclusão em um portal.

Os módulos tradicionais de programação possuem interfaces estáveis e bem definidas. A interface de um serviço *web*, entretanto, pode muitas vezes ser controlada por terceiros e assim mudar a qualquer momento. Por este motivo o projeto destaca a importância de clientes dinâmicos para serviços *web*.

3.3.2.2 Mindreef SOAPscope

Ferramenta para testar serviços *web* e diagnosticar problemas, desenvolvido pela empresa *Mindreef* (Mindreef, 2006). Suas características são:

- Permite ver documentos WSDL e mensagens SOAP na forma de pseudo-código.
- Geração automática de formulários para a invocação de serviço.
- Possibilita ver e filtrar o tráfego das mensagens SOAP.
- Integração com outras ferramentas da mesma empresa.

O resultado final é bem semelhante ao da abordagem proposta, exceto pelo fato que esta ferramenta é voltada para desenvolvedores, permitindo que eles possam testar rapidamente serviços e analisar o tráfego de mensagens. A proposta desta dissertação, por outro lado, é voltada a usuários que apenas desejam consumir serviços.

3.3.2.3 WebSphere Portlet Factory

O *WebSphere Portlet Factory* (IBM, 2006b) possui ferramentas e tecnologias para a criação, configuração, instalação e manutenção de *portlets* para o servidor de portais *WebSphere* (IBM, 2006a). Ele inclui uma ferramenta gráfica para criar, visualizar e executar *portlets*. Esta ferramenta também permite a integração de aplicações, dados e sistemas já existentes, através de componentes de geração

de código chamados *builders*. Um destes *builders*, chamado de *Service Consumer Builder*, permite criar um vínculo com um serviço pela inspeção de suas operações, permitindo a geração automática de componentes de interface de usuário. O código gerado pode ser modificado conforme a necessidade, e o *portlet* gerado pode ser instalado no portal.

Assim como a abordagem proposta por esta dissertação, o *WebSphere Portlet Factory* permite gerar automaticamente a interface de usuário de um serviço. Entretanto, esta geração não é dinâmica, mas sim feita em forma de código, que precisa de posterior compilação e instalação. Por outro lado, esta abordagem pode dar um pouco mais de flexibilidade ao desenvolvedor na hora de gerar a interface de usuário do serviço.

3.4 Conclusões

Por mais que se tente ter um modelo conceitual independente de tecnologias, a decisão de utilizar serviços *web* acaba prendendo o modelo a algumas delas. O modelo proposto se baseia nas tecnologias atuais para serviços *web*, que já estão consolidadas.

Este modelo também limitou alguns aspectos, como o suporte somente a serviços sobre HTTP, que atualmente é o protocolo de transferência usado na quase totalidade dos serviços. Entretanto o modelo pode ser facilmente adaptado para outros transportes, adaptando os rótulos que são adicionados aos vértices do grafo de serviços, apresentado na seção 3.2.3.

Em relação aos trabalhos relacionados, a maioria deles tenta facilitar a disponibilização de serviços a usuários através de técnicas de geração automática de código. Estas técnicas, apesar de apresentarem uma maior flexibilidade para o desenvolvedor (pode-se efetuar alterações no código), não permitem o dinamismo proposto por esta dissertação na hora de integrar os serviços aos portais, pois o código gerado precisa ser compilado e o componente gerado implantado no mesmo.

O próximo capítulo apresenta detalhes do protótipo construído segundo o modelo aqui proposto, e considerações que precisaram ser tomadas devido a algumas limitações práticas.

Capítulo 4

Protótipo

Este capítulo descreve como foi desenvolvido o protótipo de *software* que implementa o modelo descrito no capítulo 3. Este protótipo é, como descrito por Schach (2004), um protótipo de prova de conceito, que é construído para se estabelecer a viabilidade, limitações e direções de um projeto de *software*. Serão apresentados detalhes de sua arquitetura e funcionamento, escolhas feitas durante o desenvolvimento, e uma avaliação da sua funcionalidade.

O ambiente base para a implementação do protótipo foi o Java, versão 5.0 (Java, 2004), e bibliotecas de terceiros foram utilizadas para processar WSDL e *XML Schema*, para invocar serviços *web* e para criar o *portlet* que integra o protótipo em portais. Java foi escolhida pela grande disponibilidade de bibliotecas e ambientes de programação de alto nível de qualidade, e principalmente porque a especificação de *portlets* JSR-168 (Abdelnur e Hepper, 2003) é baseada nesta linguagem. Pelo *portlet* são servidas páginas JSP, que foram usadas para a gerar as interfaces de usuário. JSP é a tecnologia padrão Java para a geração de páginas dinâmicas em linguagem de marcação. A interface de invocação dinâmica entretanto é gerada por transformações XSL. O ambiente de desenvolvimento utilizado foi o *Eclipse*, versão 3.1 (Eclipse, 2005), com *plug-ins* para desenvolvimento de aplicações com tecnologias *web*.

O protótipo foi desenvolvido como *software livre*, sob a licença LGPL da GNU (GNU, 1999). Ele pode ser obtido no *site* `http://soap-portlet.sourceforge.net`, na forma de um pacote WAR (*Web Archive*), para ser integrado diretamente em qualquer portal compatível com a especificação de *portlets* para Java (Abdelnur e Hepper, 2003). Seu código fonte está disponível no repositório de CVS do projeto, e pode ser usado de acordo com as instruções da licença. Desde a disponibilização (março de 2006) até o momento de entrega desta dissertação o protótipo já obteve 385 *downloads* (SourceForge.net, 2006).

4.1 Detalhes da implementação dos módulos

O código fonte Java está dividido em quatro pacotes: *portlet*, *uddi*, *wsdl* e *schema*, cada um deles detalhado em uma seção a seguir. A geração dinâmica do formulário de invocação do serviço é feita

com o uso de XSL, da forma descrita a seguir. Uma estrutura de dados é criada para representar a descrição WSDL e o XML *Schema* associado. Esta estrutura é convertida em um documento XML, que representa o *grafo de esquema completo* apresentado na seção 3.2.5. Esta estrutura é passada a um transformador XSL juntamente com um *template* que implementa o algoritmo de geração de interface de usuário, apresentado na seção 3.2.6.1. O resultado é a interface em HTML. A figura 4.1 ilustra este processo, que será detalhado na seção 4.1.5.

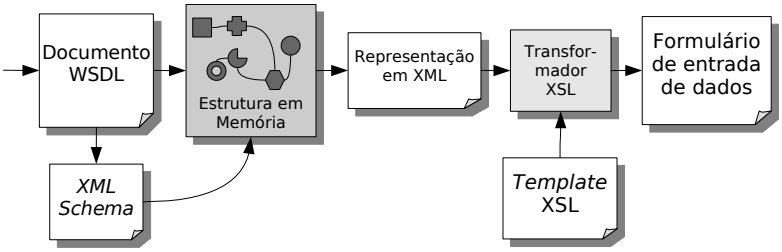

Figura 4.1: Processo de geração dinâmica de interface de usuário para invocação de serviço

A invocação dinâmica é feita pela criação de um envelope SOAP a partir dos dados inseridos pelo usuário, que é usado para a invocação do serviço, conforme detalhado na seção 3.2.7 . O envelope contendo a resposta passa por um transformador XSL, que através de outro *template* gera código HTML para apresentar o resultado da invocação ao usuário. Existe um template genérico para fazer isto, mas também é possível que o usuário forneça um *template* específico para cada serviço. O processo descrito anteriormente é mostrado na figura 4.2, e será detalhado na seção 4.1.6.

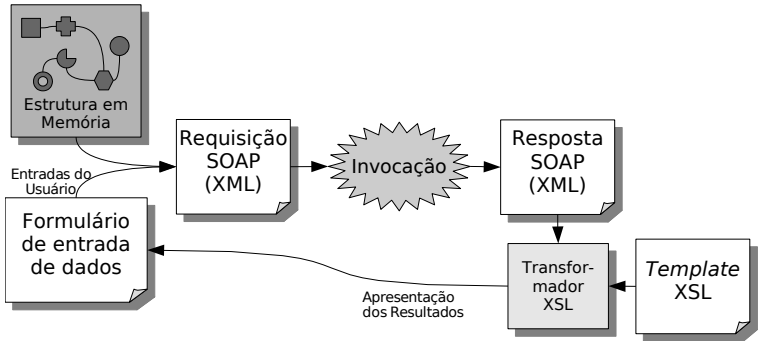

Figura 4.2: Processo de invocação dinâmica de serviço e apresentação de resultado

4.1.1 Pacote portlet

O protótipo foi construído como um *portlet*, correspondendo ao módulo descrito na seção 3.2.1. A criação de um *portlet* de acordo com a especificação de *portlets* para Java (Abdelnur e Hepper, 2003) demanda a criação de uma classe que implemente a interface *Portlet*, do pacote *javax.portlet*. Como conveniência a API oferece a classe abstrata *GenericPortlet*, que implementa boa parte dos métodos de *Portlet*, permitindo que o desenvolvedor estenda esta classe e implemente apenas os métodos restantes, relativos às requisições que o *portlet* recebe. Para este protótipo foi criada a classe *SoapClientPortlet*, que estende *GenericPortlet*, como mostrado pelo diagrama de classes na figura 4.3.

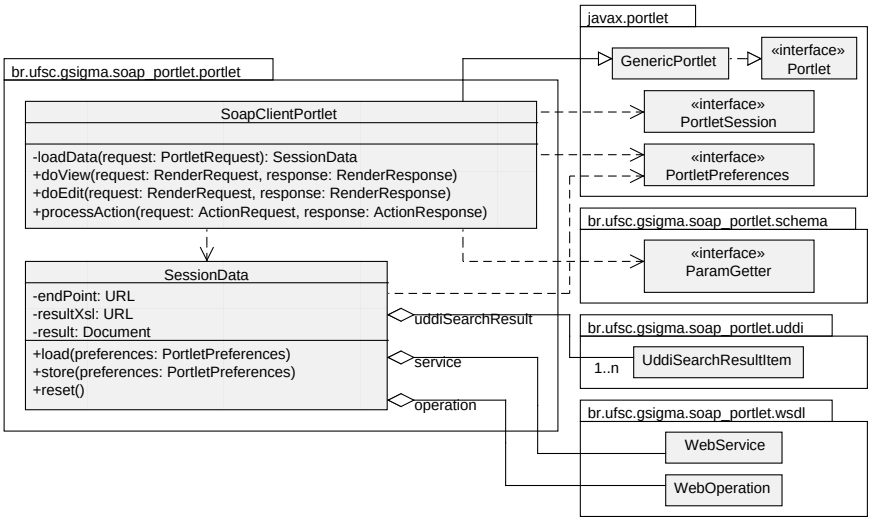

Figura 4.3: Diagrama UML do pacote “portlet”

A interface de usuário do *portlet* é gerada por duas páginas JSP. Uma delas provê o modo de configuração, onde o usuário pode configurar o serviço a ser invocado. A outra provê o modo de invocação, cujos componentes de entrada de dados são gerados dinamicamente por transformações XSL, como será explicado na seção 4.1.5. A comunicação entre estas páginas e o *portlet* é feita pelo uso de sessões, que são representadas por objetos do tipo *PortletSession*. Os dados da sessão são concentrados na classe *SessionData*, que contém:

- **uddiSearchResult:** Lista de resultados da última busca UDDI feita pelo usuário.
- **service:** Serviço selecionado pelo usuário.
- **operation:** Operação do serviço selecionada pelo usuário.
- **endPoint:** Endereço de invocação do serviço, que pode ser diferente do especificado na WSDL do serviço.
- **result:** Envelope SOAP contendo a última invocação do serviço.
- **resultXsl:** Endereço do documento XSL para apresentar o resultado para o usuário. Um documento padrão é fornecido, mas é permitido que usuários forneçam um específico.

A API de *portlets* oferece ainda a classe *PortletPreferences* para armazenar configurações de forma persistente. *SessionData* possui métodos para salvar e carregar seus dados desta classe.

4.1.2 Pacote uddi

Para a pesquisa em repositórios do módulo *Pesquisador UDDI* (descrito na seção 3.2.2) foi utilizada a biblioteca *UDDI4J*, uma biblioteca de classes Java que provê uma API para interagir com registros UDDI (UDDI4J, 2005). A invocação da biblioteca foi encapsulada pela classe *UddiSearcher*, que possui uma instância de *UDDIProxy*, que é quem efetua realmente a pesquisa. A resultado de uma pesquisa é obtido na forma de uma lista de *UddiSearchResultItem*. A figura 4.4 mostra a relação entre as classes implementadas e as classes da biblioteca.

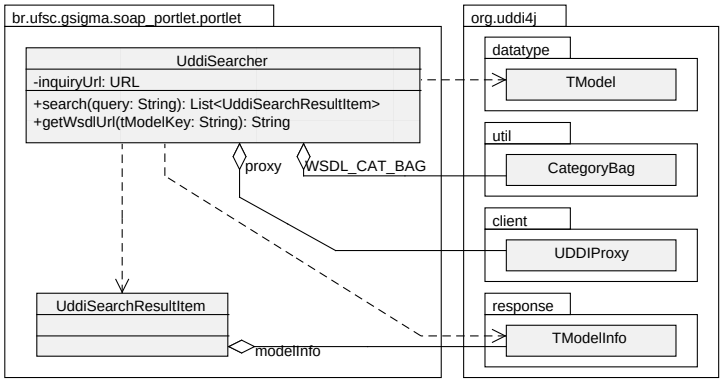

Figura 4.4: Diagrama UML do pacote “uddi”

A busca por serviços é efetuada como descrito na seção 3.2.2, ou seja, são procurados elementos do tipo *tModel*, que representem descrições WSDL. Para tanto, um *CategoryBag* é enviado junto na pesquisa, para que apenas sejam obtidos os resultados que interessam.

Os resultados da busca não contêm os *links* para os documentos WSDL, apenas as chaves para os elementos *tModel* que os contêm. A WSDL é obtida pelo método *getWsdlUrl*, que recebe como entrada esta chave.

4.1.3 Pacote wsdl

O processamento de documentos WSDL é feito pela biblioteca *WSDL4J*, que permite a criação, representação e manipulação de documentos WSDL (WSDL4J, 2005). Esta biblioteca é uma implementação de referência da *Java API for WSDL* (Duftler et al., 2005), que é uma API para a representação de documentos WSDL em Java.

Duas classes compõem este pacote: *WebService* e *WebOperation*, representando respectivamente um serviço *web* e uma operação de serviço *web*. Estas classes representam o *Grafo de Serviços*, que foi apresentado na seção 3.2.3. Elas e suas dependências são mostrados na figura 4.5.

Quando um usuário configura uma WSDL no *portlet*, via busca UDDI ou indicação direta do URL do descritor, uma instância de *WebService* é criada. Este objeto usa a classe *WSDLFactory* para criar uma instância de *WSDLReader* (ambas da biblioteca WSDL4J) para processar o documento

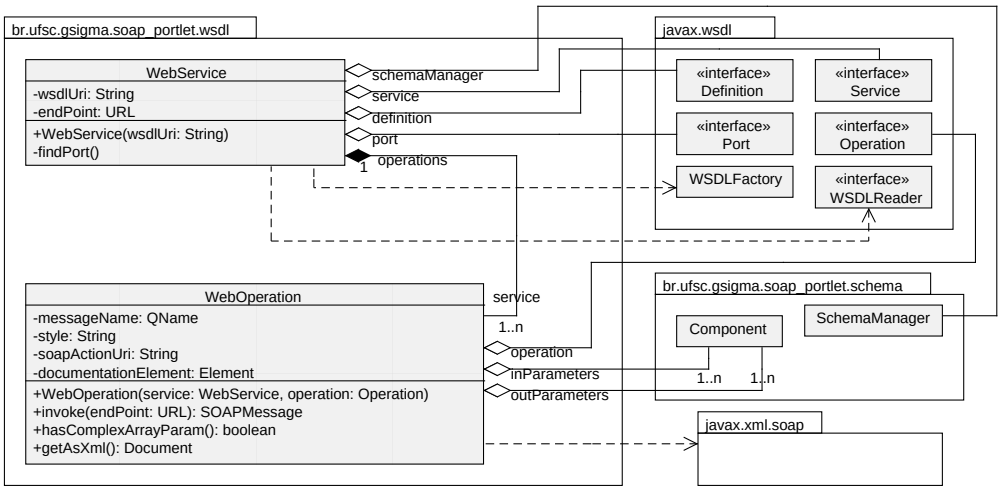

Figura 4.5: Diagrama UML do pacote “wsdl”

WSDL. Deste processamento resulta uma estrutura de objetos que replica em memória a estrutura do documento WSDL, representada por uma instância de *Definition*. O processamento da informação *XML Schema* contida na estrutura é delegado para um *SchemaManager*, do pacote *schema*.

Com as demais informações é criada uma lista de instâncias de *WebOperation*, cada uma com os parâmetros de entrada e saída da operação. Este parâmetros são representados por instâncias da classe *Component*, que são geradas pelo processamento do *XML Schema*. Isto pode ser feito de duas formas diferentes, dependendo do estilo da operação:

- **document:** Os parâmetros da operação referenciam diretamente elementos XML. Portanto, o *Component* que representa este elemento é associado ao parâmetro.
- **rpc:** Os parâmetros da operação referenciam tipos de elemento. Neste caso para cada parâmetro é criado um novo *Component*, com o tipo específico.

A classe *WebOperation* possui ainda métodos para gerar a representação XML da operação representada e para invocá-la. Ambos os métodos delegam parte de sua implementação às classes que representam os tipos de dados. O processo completo será mostrado nas próximas seções.

4.1.4 Pacote schema

O processamento dos XML *Schema* é feito utilizando-se componentes da biblioteca Xerces (Apache, 2004). Estes componentes implementam *API para XML Schema* (Litani, 2004).

Este é o maior dos pacotes do sistema, e suas classes representam principalmente o *Grafo de Tipos*, da seção 3.2.4. Sua classe principal é a *Component*, que representa um elemento XML. Cada *Component* possui um nome, um tipo e um valor. Tipos são representados pela classe abstrata *Type*,

que possui um nome, uma referência para a estrutura de dados que representa o tipo na API para *XML Schema* e um valor booleano que indica se o tipo representa um valor repetitível[1]. Há duas classes concretas derivadas: *SimpleType* e *ComplexType*, que representam respectivamente tipos simples e complexos, de acordo com a definição do *XML Schema* (apresentada na seção 2.1.3). Um tipo complexo é formado pela lista de elementos (*Component*) que o compõe, e um tipo simples pela lista das restrições que são aplicadas ao tipo. Nesta versão do protótipo apenas restrições de enumeração são tratadas, portanto esta lista representa os valores que o tipo simples aceita. Tipos possuem ainda um método para gerar uma representação XML de sua estrutura (método *asElement*). Esta representação é usada para a geração dinâmica da interface de usuário.

O atributo valor de um *Component* armazena o valor inserido pelo usuário no campo de entrada de dados correspondente. O comportamento de um valor é definido pela interface *Value*. Um valor pode ser lido de um componente de interface de usuário (através da interface auxiliar *ParamGetter*) ou inserido em um elemento SOAP. Há três implementações de valor (cada *Component* cria uma delas para si invocando o método *createValueHolder* de seu tipo). As classes são:

- **SingleValue:** Armazena o valor de um elemento com tipo simples não repetitível.
- **MultiValue:** Armazena os valores de um elemento com tipo simples repetitível
- **ComplexValue:** Guarda uma referência para o tipo complexo do elemento. Através desta referência os valores individuais podem ser acessados[2].

Tipos complexos repetitíveis não são suportados nesta versão do protótipo, por uma limitação na geração dos campos de entrada de dados para eles. Isto será explicado melhor na próxima seção.

Todos estes objetos são gerenciados pela classe *SchemaManager*, que corresponde ao *Gerenciador de Esquemas XML*, da seção 3.2.5. Esta classe é usada pela classe *WebOperation* para criar os elementos que representam seus parâmetros. Elementos fazem uso desta classe para criar seus tipos, que por sua vez a usam para criar seus elementos (caso sejam tipos complexos). A figura 4.6 apresenta o diagrama das classes deste pacote.

4.1.5 Geração de interface de usuário

O algoritmo de geração dinâmica de interfaces de usuário, introduzido na seção 3.2.6.1, foi implementado como um *template* XSL. Isto proporcionou a separação completa entre o código de linguagem de marcação, que representa a interface de usuário, e o código fonte do *portlet*. Esta característica é importante, pois permite a adaptação do *portlet* para o uso de linguagens de marcação diferentes sem que haja necessidade de alterações profundas. O uso de *templates* XSL na implementação do protótipo foi influenciado pelo trabalho de Fiorini (2006), citado na seção 3.3.1.2, que os usava para apresentar conteúdo XML para clientes com suporte a diferentes linguagens de marcação.

[1] Para cada tipo, a versão repetitível é diferenciada da não repetitível.

[2] Isto é possível porque instâncias das classes de tipo não são compartilhadas.

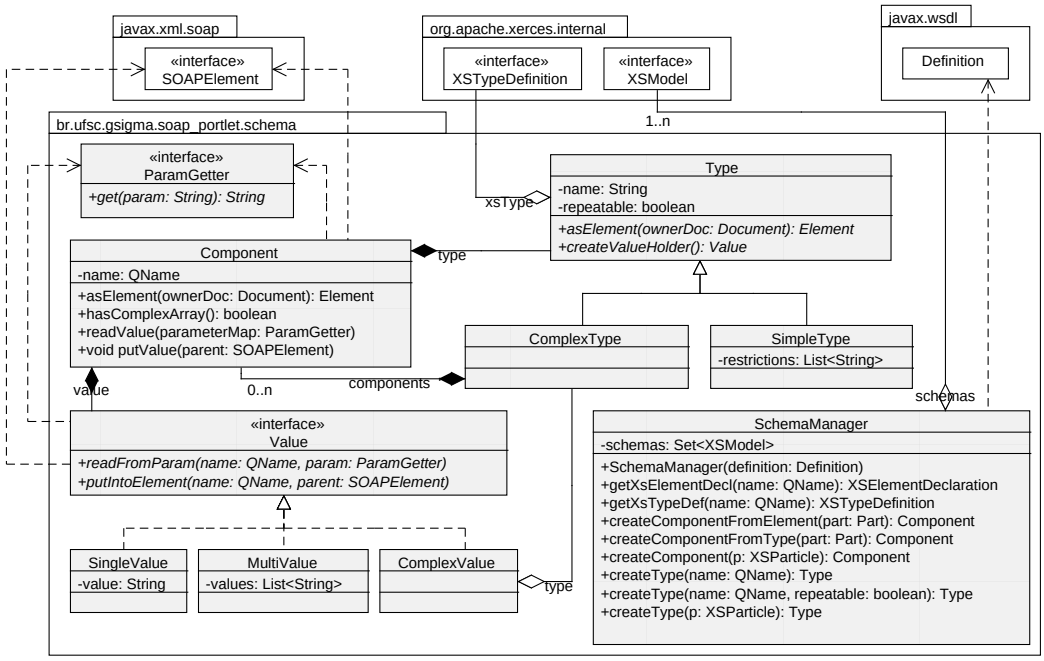

Figura 4.6: Diagrama UML do pacote “schema”

4.1.5.1 Geração da representação XML

Para que se possa usar XSL é necessário ter-se uma representação em XML da operação que se deseja invocar e das estruturas de dados usadas pelos parâmetros. Para tanto, a classe *WebOperation* possui o método *getAsXml*, responsável por gerar esta representação em XML. Ele funciona como mostrado no algoritmo 4.1. Parte da implementação é delegada às classes que representam os tipos de dado (*Type* e suas derivadas). Cada tipo gera uma representação de si em XML pelo método *asElement*. Em tipos simples, como mostrado no algoritmo 4.2, são inseridas informações de repetição ou restrições de enumeração. Em tipos complexos, são representados seus elementos, cujas representações de tipo são recursivamente geradas, como mostrado pelo algoritmo 4.3. O documento é gerado de acordo com o *XML Schema* apresentado no apêndice A.

Algoritmo 4.1 Geração da representação XML de uma operação

1: seja *o* a operação
2: gerar um elemento *operation*, com o nome de *o* como atributo
3: **para cada** parâmetro de entrada *p* de *o* **faça**
4: gerar um elemento *param*, com o nome de *p* como atributo
5: inserir no elemento *operation* o elemento *param* recém criado
6: inserir no elemento *param* a representação XML do tipo de *p*
7: **fim para**

A figura 4.7 mostra um diagrama de objetos representando uma operação de um serviço de *Previsão de Tempo* com dois parâmetros: o *lugar* sobre o qual se deseja saber a previsão, e o *formato* desta

Algoritmo 4.2 Geração da representação XML de um tipo simples

```
1: seja t o tipo simples
2: gerar um elemento simple-type, com o nome de t como atributo
3: se t for repetitível então
4:     colocar o atributo repeat="yes" no elemento simple-type
5: fim se
6: se t possuir restrições de enumeração então
7:     colocar o atributo enum="yes" no elemento simple-type
8:     para cada valor v pertencente aos possíveis valores de t faça
9:         gerar um elemento enum-value, com v como conteúdo
10:        inserir o elemento enum-value criado em simple-type
11:    fim para
12: fim se
```

Algoritmo 4.3 Geração da representação XML de um tipo complexo

```
1: seja t o tipo complexo
2: gerar um elemento complex-type, com o nome de t como atributo
3: para cada elemento e de t faça
4:     gerar um elemento element, com o nome de e como atributo
5:     inserir no elemento complex-type o elemento element recém criado
6:     inserir no elemento element a representação XML do tipo de e
7: fim para
```

previsão. O *lugar* é um parâmetro composto, com dois elementos: o *país*, e suas *cidades* sobre as quais se deseja obter a previsão. *País* é do tipo *string*, assim como *cidade*, mas este último pode ser repetido, ou seja, pode-se obter a previsão para diversas cidades de um mesmo país. O *formato* é um parâmetro simples, cujo tipo pode assumir apenas os valores *Resumida* e *Completa* (este tipo possui uma restrição de enumeração).

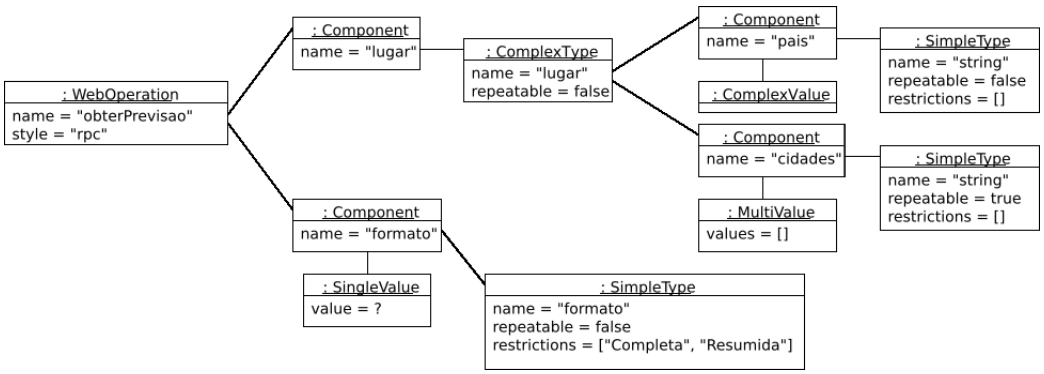

Figura 4.7: Diagrama de objetos de uma operação de um serviço de previsão do tempo

A invocação do método *getAsXml* executará os algoritmos apresentados anteriormente, gerando a representação XML apresentada na figura 4.8.

```
<?xml version="1.0" encoding="UTF-8"?>
<operation name="obterPrevisao" service="PrevisaoService">
  <param name="lugar">
    <complex-type name="lugar">
      <element name="pais">
        <simple-type name="string" />
      </element>
      <element name="cidades">
        <simple-type name="string" repeat="yes" />
      </element>
    </complex-type>
  </param>
  <param name="formato">
    <simple-type enum="yes" name="formato">
      <enum-value>Completa</enum-value>
      <enum-value>Resumida</enum-value>
    </simple-type>
  </param>
</operation>
```

Figura 4.8: Descrição XML de uma operação de serviço

4.1.5.2 Transformação XSL

O *template* XSL[3] processa a representação XML previamente descrita da seguinte forma:

- **Elemento *operation*:** Este é o elemento raiz da representação XML. Para ele é gerada uma tabela, cujo conteúdo será gerado pelo processamento de seus elementos filhos, e o botão que o usuário usará para invocar o serviço.
- **Elementos *param*:** Para cada um destes elementos é gerada uma linha de tabela. Nesta linha, a primeira célula contém o nome do parâmetro, e as demais serão geradas pelo processamento do elemento filho.
- **Elementos *simple-type*:** Para estes elementos é gerada uma célula com um elemento de formulário para entrada de texto.
- **Elementos *simple-type* com *repeat="yes"*:** Para estes elementos é gerada uma célula com um elemento de formulário para entrada de texto com capacidade para múltiplas linhas. Para a invocação do serviço, cada linha será considerada uma instância do elemento repetitível.
- **Elementos *simple-type* com *enum="yes"*:** Para estes elementos é gerada uma célula com um elemento de formulário do tipo lista de escolha. Cada um dos possíveis valores do tipo será um item da lista.
- **Elementos *complex-type*:** Para estes elementos é gerada uma célula contendo uma outra tabela aninhada. As linhas desta tabela são geradas pelo processamento dos elementos filhos.

[3] Mostrado integralmente no apêndice B.

- **Elementos *element*:** De forma análoga aos elementos *param*, o processamento destes elementos gera uma linha de tabela, cuja primeira célula contém o nome do elemento, e as demais serão geradas pelo processamento do elemento filho que representa o tipo.

Cada um dos componentes de formulário recebe um identificador para que o dado inserido pelo usuário possa ser obtido posteriormente. Componentes que representam parâmetros de tipo simples são identificados com o próprio nome do parâmetro. No caso de componentes que são elementos de um parâmetro de tipo complexo, o identificador será a concatenação do nome do parâmetro com o nome do elemento, separados por um caractere especial. Casos mais complexos seguem a mesma regra: seus nomes serão a concatenação de todos os elementos na hierarquia, do parâmetro até o componente de formulário.

Como descrito, elementos com tipos simples repetitíveis possuem campos de texto com múltiplas linhas para a entrada de dados. Através disso é possível a um usuário passar o número de parâmetros desejado. Para tipos complexos repetitíveis isto não é possível, pois são formados por mais de um dado de entrada. Por exemplo, um parâmetro repetitível do tipo *string* pode ser preenchido por um usuário em um campo de texto de múltiplas linhas. A cada nova linha de texto entrada pelo usuário um novo elemento é considerado. Entretanto, para um parâmetro repetitível do tipo *Pessoa* isto não é possível (assumindo que este tipo seja complexo, contendo os elementos nome e idade, do tipo *string* e *integer* respectivamente). Caso o usuário queira entrar com uma nova pessoa, todos os seus campos de entrada de dados precisam ser repetidos, de forma semelhante ao que é mostrado na figura 4.9. Esta limitação poderia ser superada com o uso de uma linguagem de *script* que rode no navegador do usuário (*Javascript*), que poderia gerar dinamicamente os campos extra.

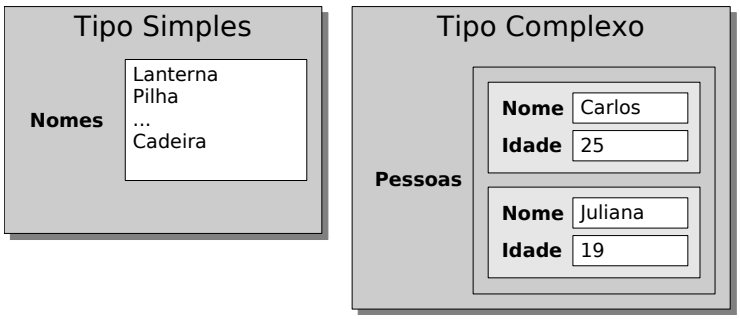

Figura 4.9: Diferença na apresentação de tipos repetitíveis simples e complexos

A figura 4.10 apresenta um diagrama de seqüência, das operações executadas para a geração da interface de usuário a partir do momento que ele passa um documento WSDL ao *portlet*. Ao receber a requisição de ação correspondente à atribuição do descritor de serviço, o *portlet* a passa a sua instância de *SessionData*, que cria um novo *WebService*. Durante sua criação, esta classe cria um *SchemaManager*, que cuida do grafo de tipos do *XML Schema* presente no descritor WSDL , procura pelo primeiro *port* SOAP declarado, e cria uma lista de *WebOperation*, uma para cada uma de suas operações. As operações, por sua vez, criam um *Component* para cada um de seus parâmetros, e utilizam o *SchemaManager* para criar um tipo para eles. Após a requisição de ação, o *portlet* recebe uma requisição de renderização. Durante seu processamento o objeto *SessionData* é carregado na

sessão do usuário e a renderização é delegada ao arquivo *view.jsp*, através do método *include*. Dentro do JSP é verificado qual operação do serviço foi escolhida pelo usuário, e sua representação em XML é finalmente gerada. A geração acontece de forma recursiva, até os tipos[4]. O documento resultante é transformado então, com o auxílio do *template* XSL .

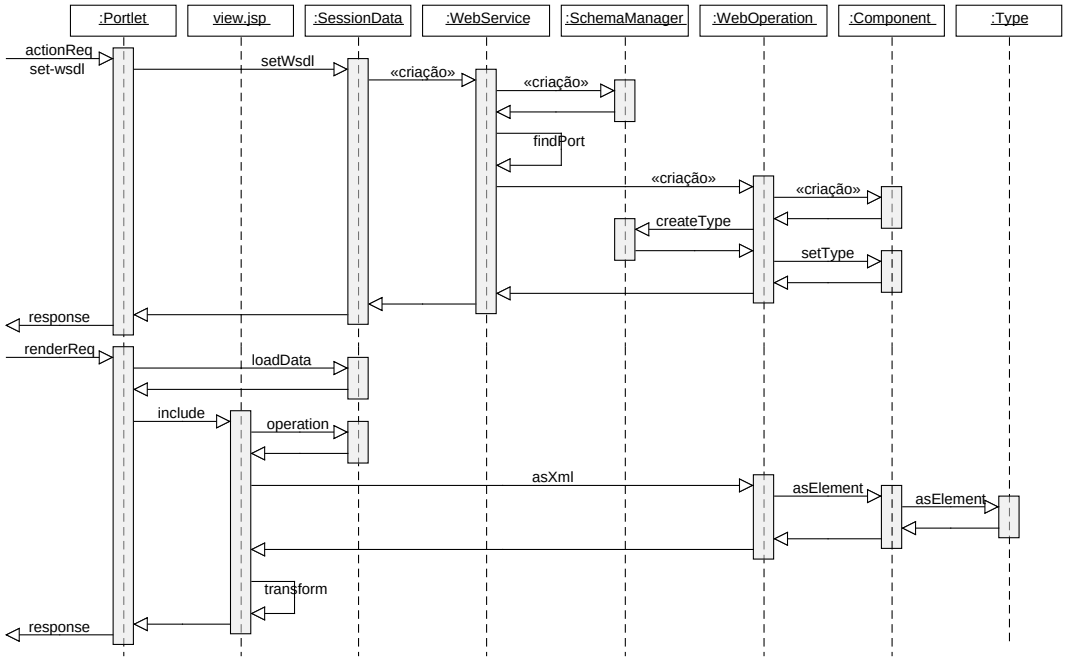

Figura 4.10: Diagrama de seqüência da geração de interface de usuário

A figura 4.11 mostra como ficaria o conteúdo HTML gerado pela transformação XSL aqui apresentada. A representação visual deste conteúdo pode ser visto na figura 4.15 (página 75). Em destaque na figura estão os nomes dos parâmetros, apresentados ao usuário e os nomes dos campos de formulário, mostrando nomes concatenados para elementos de parâmetros compostos.

4.1.6 Invocação de Operações

A invocação do serviço é disparada pelo usuário quando ele pressiona o botão gerado pelo *template*. Durante o processamento da requisição de ação, no *portlet*, os parâmetros da requisição (que contêm os valores inseridos pelo usuário nos componentes de formulário, cada um associado ao identificador do respectivo componente) são passados ao objeto que representa a operação. Isto é feito pela invocação do método *readValue* de cada um dos parâmetros da operação, que faz com que o método *readFromParam* de seu objeto *Value* seja invocado. Este método obtém o valor do conjunto de parâmetros da requisição através do nome e o armazena. Para os tipos complexos o método faz isso recursivamente pelos elementos do tipo.

[4]O diagrama apresenta a invocação de *asElement* diretamente na interface *Type*, mas na realidade ela é feita em instâncias das classes que a implementam: *SimpleType* e *ComplexType*.

```
<h1>obterPrevisao</h1>
<form action="POST">
  <table>
    <tr>
      <td><strong>lugar</strong></td>
      <td colspan="2">
        <table>
          <tr>
            <td><strong>pais</strong></td>
            <td><input name="param/lugar/pais"></td>
            <td><em>string</em></td>
          </tr>
          <tr>
            <td><strong>cidades</strong></td>
            <td><textarea name="param/lugar/cidades"></textarea></td>
            <td><em>List of string</em></td>
          </tr>
        </table>
      </td>
    </tr>
    <tr>
      <td><strong>formato</strong></td>
      <td>
        <select name="param/formato">
          <option value="Completa">Completa</option>
          <option value="Resumida">Resumida</option>
        </select>
      </td>
      <td><em>formato</em></td>
    </tr>
  </table>
  <p><input value="Invoke!" type="submit"></p>
</form>
```

Figura 4.11: Conteúdo HTML gerado pela transformação XSL

Após os parâmetros serem lidos, a operação é invocada, através do método *invoke*. A execução deste método cria um envelope SOAP usando a API SAAJ (Jayanti e Hadley, 2006), provida pela biblioteca Axis, do projeto Apache (Apache, 2005). No elemento *body* deste envelope é inserido o elemento contendo o nome da operação invocada, e neste são inseridos valores dos parâmetros da operação. Para tanto, a classe *Component* possui o método *putValue* para inserir seu valor em um elemento SOAP. Este método invoca o método *putIntoElement* do objeto *Value* do parâmetro.

Elementos de parâmetros complexos são tratados recursivamente, inserindo-se nos elementos-pais correspondentes. Após o término da montagem do envelope, este é enviado ao ponto de acesso do serviço, que retribui com o envelope SOAP contendo a resposta. A resposta é armazenada na sessão do usuário, e na próxima requisição de renderização ela será transformada em linguagem de marcação. A figura 4.12 apresenta o diagrama de seqüência deste processo e a figura 4.13 mostra como fica o envelope SOAP gerado a partir do diagrama de objetos da figura 4.7 e dos parâmetros inseridos pelo usuário.

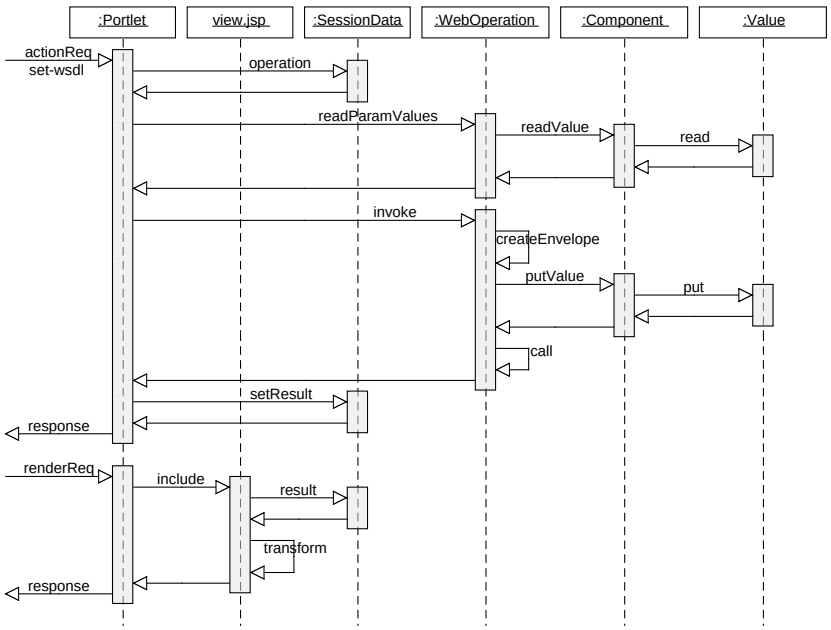

Figura 4.12: Diagrama de seqüência da invocação de um serviço e da apresentação de seu resultado

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <obterPrevisao xmlns="urn://previsao">
      <lugar>
        <pais>Brasil</pais>
        <cidades>Florianópolis</cidades>
        <cidades>Blumenau</cidades>
        <cidades>Manaus</cidades>
      </lugar>
      <formato>Resumida</formato>
    </obterPrevisao>
  </soapenv:Body>
</soapenv:Envelope>
```

Figura 4.13: Envelope SOAP gerado para a invocação do serviço

4.1.7 Telas

Aqui serão mostradas duas capturas de tela do *portlet*, rodando no servidor de portais *Jetspeed 2*, da Apache (Apache, 2006), a fim de dar uma idéia simplificada da utilização da abordagem proposta. A figura 4.14 mostra uma tela de portal com duas instâncias do *portlet*. A primeira configurada para invocar um serviço de dicionário, e a segunda com um serviço de informações sobre países, o qual já foi invocado.

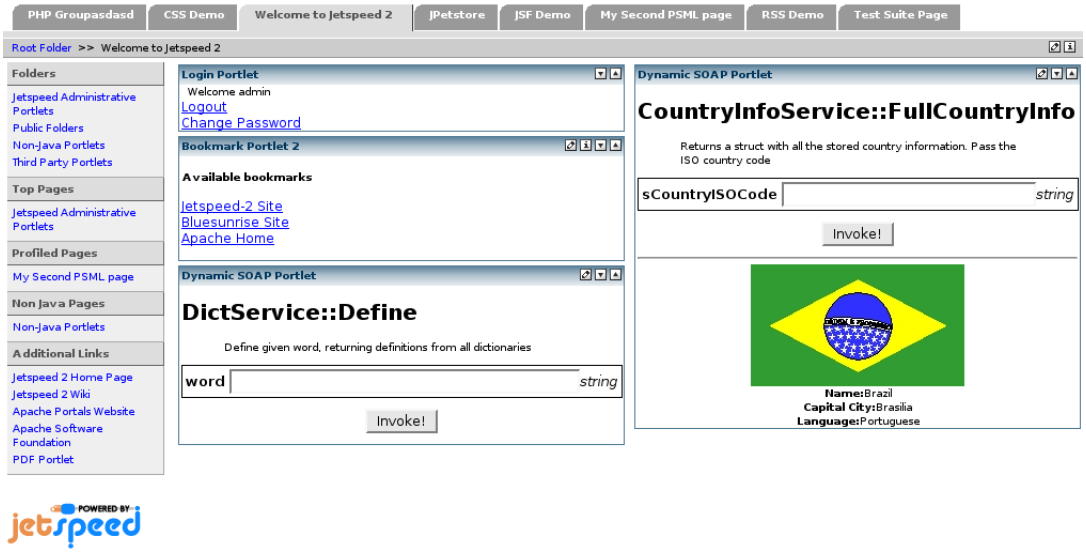

Figura 4.14: Portal com duas instâncias do *portlet*

4.1.7.1 Modo de Configuração

A figura 4.15 mostra uma captura de tela do *portlet*, em seu modo de configuração. Seu primeiro formulário permite fazer pesquisa em um registro UDDI, do qual se falará a seguir. O próximo formulário pode ser preenchido manualmente para se indicar o endereço do arquivo WSDL de um serviço. A opção *Invocation Endpoint*, que permite a indicação do ponto de acesso do serviço, é preenchida com dados da WSDL, mas pode ser preenchida manualmente também. Em seguida são apresentados as operações disponíveis pelo serviço da WSDL selecionada e seus parâmetros. Em serviços com mais de uma operação há uma lista de seleção para a escolha de qual operação usar. O último formulário permite a indicação opcional do endereço de um arquivo XSL, para formatar a resposta do serviço. Este exemplo é de um serviço de previsão do tempo, o mesmo apresentado na figura fig:exemplo-desc-xml-simpl. Pode-se observar na figura a listagem dos parâmetros lá declarados.

A figura 4.16 mostra o resultado da busca no repositório UDDI por "*weather*", com os resultados logo abaixo. Selecionando-se um deles, o URL de seu descritor WSDL é automaticamente associado ao *portlet*.

4.1.7.2 Modo de Invocação

Na figura 4.17 é mostrada uma captura de tela do *portlet* em modo de invocação, já com um serviço configurado para o mesmo exemplo de serviço do diagrama de objetos apresentado na figura 4.7, e após uma invocação já ter sido feita. Na parte superior da tela estão os campos de formulário gerados dinamicamente para a operação selecionada, e o botão de invocação. A parte inferior mostra o resultado da última invocação. Aqui o resultado está sendo mostrado sem folha de estilo, e portanto

Figura 4.15: *Portlet* em modo de configuração

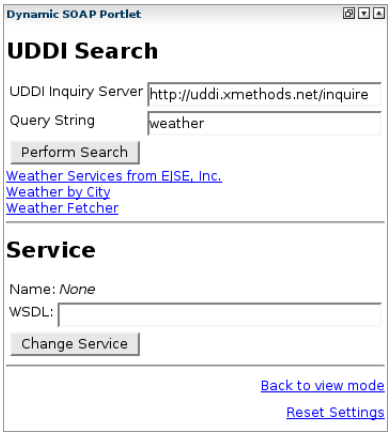

Figura 4.16: Busca em UDDI

apenas a estrutura hierárquica da resposta é apresentada. Foram retornadas as previsões para as três cidades requeridas na invocação.

4.2 Avaliação do Protótipo

Para que se pudesse avaliar a funcionalidade do protótipo desenvolvido, bem como identificar eventuais limitações, uma bateria de testes com diversos serviços *web* foi feita para avaliar suas capacidades práticas. Estes testes, e as conclusões obtidas, são apresentados na seção 4.2.1.

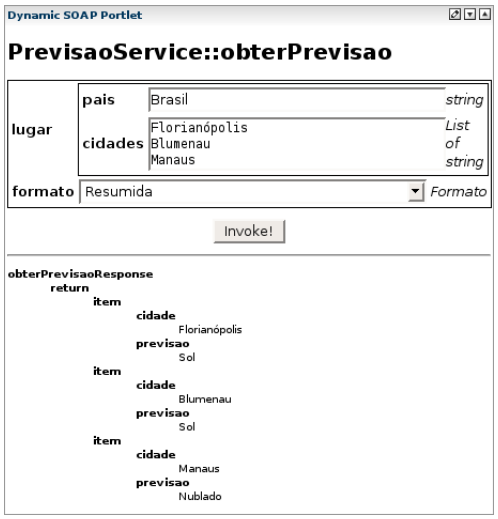

Figura 4.17: *Portlet* em modo de invocação

Além disto, a seção 4.2.2 apresenta algumas considerações sobre as vantagens do uso da abordagem apresentada em relação a abordagens tradicionais para integração de serviços web em portais.

4.2.1 Testes

Para a realização dos testes foram usados serviços disponíveis publicamente no *site* XMethods (2004), que contém cerca de quinhentos serviços cadastrados. Destes serviços, dez por cento foram escolhidos ao acaso[5]para se testar o protótipo. Os resultados são apresentados de forma resumida na tabela 4.1, e em detalhes no apêndice C.

Resultado	Quantidade
Erro ao processar WSDL	3
Impossível invocar por necessitar cadastro	7
Impossível invocar por falta de informações	6
Serviço de baixo nível	2
Erro ao invocar	2
Invocação com sucesso	30
Total	*50*

Tabela 4.1: Resumo dos testes

A biblioteca usada para processar arquivos WSDL (WSDL4J) não conseguiu processar o descritor de três serviços. Para todos os demais este processamento foi possível, e pode-se configurar o *portlet* para gerar a interface de usuário para invocar suas operações. Destes, sete requeriam um cadastro prévio no servidor para que pudessem ser invocados, e não foram testados. Outros seis não continham documentação suficiente sobre seus parâmetros, ou exigiam parâmetros de difícil preenchimento, como por exemplo códigos postais do Canadá. Outros dois serviços (serviços de baixo nível) não

[5]Para tanto foi criado um *script* que obtinha a lista dos serviços e a embaralhava. Os cinqüenta primeiros itens desta lista foram utilizados nos testes.

permitiam a invocação pela interface gerada, pois esperavam dados codificados em forma binária[6]. Dos restantes, trinta foram invocados com sucesso, ou seja, o servidor retornou um envelope SOAP contendo dados relevantes, e não erros. Dois serviços entretanto apresentaram erros. Em um deles o servidor não respondia, e em outro o formato de mensagem enviado não era entendido.

4.2.1.1 Avaliação dos testes

O espaço amostral foi constituído de serviços bem variados e de diferentes provedores, o que permitiu tirar algumas conclusões. Primeiramente, o algoritmo para a geração de interfaces de usuário se mostrou bem robusto. Com a exceção dos casos onde a biblioteca utilizada não conseguiu processar os descritores WSDL, para todos os serviços foram gerados campos de entrada de dados para seus parâmetros, da forma esperada.

Para quinze serviços foi possível gerar a interface de usuário, mas não testar a invocação. Alguns serviços necessitavam cadastro, seja na forma do um parâmetro extra contendo uma chave, ou um desbloqueio do endereço IP do invocador. Outros serviços ainda não puderam ser invocados por falta de informações para fazê-lo, e houve também casos de serviços que não eram suportados pela abordagem proposta.

Dos serviços que puderam ser invocados, dois apresentaram problemas. Em um deles o servidor não respondia à invocação, o que caracteriza um problema do servidor. O outro não conseguia entender a mensagem enviada. Nos outros trinta a invocação se deu de forma natural. Este fato permite deduzir que o formato de envelope gerado está em conformidade com o que a grande maioria dos serviços espera.

4.2.2 Considerações em relação a outras abordagens

Com os testes do protótipo pôde-se ver que a abordagem proposta possui grande utilidade. A maioria dos serviços testados puderam ser utilizados sem a necessidade de se programar um cliente para invocá-los. As seguintes vantagens puderam ser observadas:

- Não há a necessidade de se desenvolver um cliente específico para os serviços. Abordagens tradicionais geralmente utilizam-se de ferramentas de geração de código, que geram *proxies* automaticamente para a invocação dos serviços como se fossem objetos executando localmente, para a geração destes clientes.
- A interface de usuário para invocar serviços é gerada automática e dinamicamente conforme a definição de cada serviço. Não é preciso se preocupar em ler o descritor do serviço (o que geralmente é complicado) ou as assinaturas dos métodos dos *proxies* gerados para saber quais parâmetros um serviço requer, projetar a interface de usuário para estes parâmetros e finalmente implementá-la.

[6]Como descrito na seção 3.1, este tipo de serviço não é suportado pela abordagem.

- O código para invocar o serviço através do *proxy* gerado, passando os parâmetros inseridos pelo usuário, também não precisa ser escrito. O mesmo é válido para a apresentação dos resultados do serviço.

- Não há a necessidade de se desenvolver um *portlet* para integrar serviços em portais, nem de um profissional especializado. O protótipo foi desenvolvido diretamente como um *portlet*, de forma que pode ser instalado diretamente em servidores de portal compatíveis. Com isto não é preciso conhecer APIs para este fim. De acordo com a experiência pessoal do autor o tempo economizado com o desenvolvimento de *portlets* para invocar um serviço específico, por um profissional com amplo conhecimento nas tecnologias empregadas e sem utilizar nenhuma ferramenta auxiliar, é de cerca de uma a duas horas. Considerando que um número grande de serviços o tempo e os custos com treinamento passam a ser consideráveis

- Como não há escrita e compilação de código, o *portlet* só precisa ser instalado uma vez no servidor de portal de destino, e diversos serviços diferentes podem ser utilizados, bastando apenas alterar as configurações.

- Este modelo promove uma separação bem clara entre lógica e apresentação em aplicações para portais, com a lógica implementada na forma de um serviço *web*, e a apresentação sendo fornecida pelo *portlet* do protótipo. Mudanças na lógica, que impliquem em alteração na interface do serviço, refletem-se diretamente no *portlet*, onde a interface de usuário será ajustada de acordo.

Entretanto, algumas desvantagens também podem ser observadas:

- Certos serviços, como apresentado anteriormente, não são propícios para a invocação direta por usuários. Estes serviços não são suportados pelo *portlet* genérico, como já foi previsto na definição da abordagem. O máximo proveito que se poderia tirar nestes casos seria a possibilidade de testar estes serviços, para então utilizá-los de outras formas.

- A facilidade que o *portlet* genérico traz é contrabalançada pela perda de flexibilidade que o acompanha:

  - Ele não permite qualquer tipo de interação entre operações do serviço ou entre outros serviços: cada instância do *portlet* permite invocar apenas uma operação do serviço por vez. Isto é um problema grave em serviços que necessitem obrigatoriamente de chamadas múltiplas (por exemplo, uma inicial para obter um identificador de sessão, que deve ser enviado em chamadas subseqüentes). O conceito de *Orquestração de Serviços* (Peltz, 2003) poderia ser aplicado para solucionar este problema.
  - Cada parâmetro do serviço deve obrigatoriamente ser inserido pelo usuário. Não se pode fazer com que algum parâmetro tenha um valor fixo ou calculado de alguma forma em função dos outros parâmetros. Isto pode ser um inconveniente para usuários, que precisariam sempre preencher estes valores.

– Não se pode mudar a estrutura da interface gerada. Por questão de usabilidade, poderia se querer que os componentes gerados fossem apresentados de forma diferente. Entretanto, esta é uma limitação apenas deste protótipo, que poderia ser superada com o uso de folhas de estilo.

4.3 Conclusões sobre o protótipo

Com este protótipo pôde-se avaliar e fazer considerações práticas sobre a abordagem proposta no capítulo anterior. Um dos pontos de destaque foi o uso de XSLT para a geração dos componentes de formulário da interface de usuário. Graças a isto conseguiu-se uma separação completa entre a representação dos parâmetros e o código específico para os componentes de formulário. Desta forma será fácil adaptar o protótipo para gerar interfaces de usuário em outras linguagens de marcação.

Graças aos testes efetuados pode-se observar que nem todos os serviços *web* são adequados para a integração direta em portais. Há serviços cujo propósito é de muito baixo nível para isto. E há também muitos casos de serviços que requerem um encadeamento de invocações, o que não é possível pelo modelo proposto. Entretanto, há muitos serviços cujas invocações são baseadas apenas em poucos parâmetros simples (que são o foco principal da abordagem), e assim completamente passíveis de serem utilizados em conjunto com o protótipo gerado. Em outras palavras, o objetivo da proposta foi o desenvolvimento de interfaces de usuário apenas para serviços que as requerem.

Este protótipo está sendo usado no projeto internacional ECOLEAD (2004), que, entre outros objetivos, está desenvolvendo uma infraestrutura de tecnologia de informação e comunicação baseada em SOA. Este protótipo será um componente de portal disponível para os usuários da infraestrutura.

Finalmente, como dito na introdução deste capítulo, o protótipo foi tornado disponível como *software* livre, e sua utilidade pode ser inferida pelo grande número de *downloads* já efetuados desde a sua recente publicação.

Capítulo 5

Conclusões

Este trabalho propôs uma abordagem para a integração dinâmica de serviços *web* em portais, de forma que componentes de formulário para a invocação destes serviços possam ser gerados dinâmica e automaticamente.

A abordagem é constituída por um conjunto de algoritmos que interpretam um documento descritor de um serviço *web*, e através desta interpretação geram componentes de interface de usuário correspondentes aos parâmetros requisitados pelo serviço. Estes algoritmos também permitem que, após um usuário inserir dados naqueles componentes previamente gerados, o serviço seja invocado e seu resultado visualizado. Para que o objetivo de integração em portais fosse alcançado, propôs-se que os algoritmos fossem implementados dentro de um componente de interface de usuário de portais: um *portlet*.

Para avaliar a abordagem, um protótipo foi implementado e testado com um conjunto de serviços, de forma que se pôde verificar as suas capacidades. O protótipo fez uso de bibliotecas para o processamento dos descritores de serviço e para a invocação dos mesmos. Elas foram utilizadas para que o esforço de implementação se concentrasse apenas na abordagem proposta, e não no tratamento dos descritores, trabalho que já havia sido feito e validado pelos implementadores das bibliotecas. A geração dos componentes de interface de usuário foi feita por transformações XML, uma técnica que proporcionou a separação entre o código de linguagem de marcação dos componentes e o código que continha a lógica da implementação da abordagem.

O protótipo e seus testes mostraram que a abordagem proposta funciona. (Apesar de o protótipo não cobrir integralmente a abordagem). Serviços cujos parâmetros incluam *arrays* de tipos complexos não foram suportados, devido à raridade com que são encontrados na prática e a complicação que seu suporte acarretaria. O suporte completo a qualquer tipo de serviço e a outras limitações podem ser tratados em trabalhos futuros.

5.1 Sugestão de Trabalhos Futuros

Alguns pontos não foram tratados neste trabalho, seja por sua complexidade ou pela limitação de tempo. No entanto, eles são importantes e poderão dar continuidade na pesquisa para uma melhor integração de serviços *web* em portais. Algumas sugestões para trabalhos futuros são apresentadas nas próximas seções.

5.1.1 Geração de componentes de interface para elementos complexos repetitíveis

O protótipo tratou elementos simples repetitíveis como elementos de entrada de texto com suporte a múltiplas linhas, onde cada linha corresponde a uma repetição do elemento. Para os casos onde o elemento repetitível é complexo esta solução não se aplica. SQLData System (2003) tentou resolver o problema pela definição prévia do número de repetições, gerando todos os campos de uma só vez.

Uma solução mais adequada seria gerar dinamicamente mais componentes de formulário, para uma nova repetição do parâmetro, conforme a necessidade do usuário. Isto é possível usando-se uma linguagem que rode no navegador do usuário e permita manipular dinamicamente os elementos de linguagem de marcação, como *Javascript*.

5.1.2 Validação dos dados inseridos pelos usuários

Nenhuma validação é feita no conteúdo dos dados inseridos pelo usuário. Desta forma, o envio de dados alfanuméricos em um parâmetro que só aceita dados numéricos pode acontecer, e o erro só será detectado após a invocação do serviço. Em geral, os tipos de dados de cada parâmetro de um serviço são indicados em seu descritor WSDL, e esta informação poderia ser utilizada para verificar as entradas dos usuário antes de se fazer a invocação.

5.1.3 Estudo da aplicação de Ajax no protótipo

Aplicações *web* possuem, em geral, um problema de tempo de resposta. A página da aplicação precisa ser recarregada completamente a cada requisição. Uma técnica, conhecida como AJAX (*Asynchronous JavaScript and XML*, JavaScript Assíncrono e XML), vem sendo aplicada para solucionar, ou pelo menos amenizar, este problema. AJAX faz uso de tecnologias que permitem que a aplicação seja atualizada em pequenos trechos.

De acordo com a especificação de *portlets* (Abdelnur e Hepper, 2003), requisições de ação de um *portlet* fazem com que toda a página do portal seja recarregada. AJAX poderia sem empregado para que apenas o *portlet* alterado o fosse.

5.1.4 Uso de Orquestração para tratar interação entre serviços

Como analisado na seção 4.2.2, em muitos casos é preciso invocar várias operações de um serviço para fazer uso deste. Às vezes pode até ser preciso invocar mais de um serviço para a obtenção de resultados intermediários. A abordagem proposta trata apenas casos simples, com uma única operação. Técnicas de *Orquestração de Serviços* (Peltz, 2003) poderiam ser usadas para permitir maior interatividade. Em vez de uma única operação de um único serviço, várias operações de vários serviços poderiam ser configurados, juntamente com a forma como suas invocações se encadeariam. Isto daria mais flexibilidade a usuários, ao custo de uma maior complexidade.

Apêndice A

XML Schema do Descritor de Operação

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="operation">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="param" type="type"
                    minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="service" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="type">
    <xs:choice>
      <xs:element name="simple-type">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="enum-value" type="xs:string"
                        maxOccurs="unbounded" minOccurs="0" />
          </xs:sequence>
          <xs:attribute name="name" type="xs:string" use="required"/>
          <xs:attribute name="enum" type="yes-no" default="no" />
          <xs:attribute name="repeat" type="yes-no" default="no" />
        </xs:complexType>
      </xs:element>
      <xs:element name="complex-type">
        <xs:complexType>
```

```
          <xs:sequence>
            <xs:element name="element" type="type"
                        maxOccurs="unbounded" minOccurs="1"/>
          </xs:sequence>
          <xs:attribute name="name" type="xs:string" use="required"/>
        </xs:complexType>
      </xs:element>
    </xs:choice>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>

  <xs:simpleType name="yes-no">
    <xs:restriction base="xs:string">
      <xs:enumeration value="yes"/>
      <xs:enumeration value="no"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

Apêndice B

Template XSL para a Geração de Componentes de Interface de Usuário

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output indent="yes" method="html"/>
  <xsl:param name="invoke-url" />
  <xsl:template match="/operation">
    <h1>
      <xsl:value-of select="./@service"/>::
      <xsl:value-of select="./@name"/>
    </h1>
    <xsl:if test="./documentation">
      <blockquote>
        <xsl:value-of select="./documentation"/>
      </blockquote>
    </xsl:if>
    <form action="POST">
      <xsl:attribute name="action">
        <xsl:value-of select="$invoke-url"/>
      </xsl:attribute>
      <table style="border: thin solid">
        <xsl:apply-templates select="./param"/>
      </table>
      <p style="text-align: center">
        <input type="submit" value="Invoke!"/>
      </p>
    </form>
  </xsl:template>
```

```
<xsl:template match="param">
  <tr>
    <td style="vertical-align: middle">
      <strong><xsl:value-of select="./@name"/></strong>
    </td>
    <xsl:apply-templates select="./*"/>
  </tr>
</xsl:template>
<xsl:template match="simple-type[@repeat='yes']">
  <xsl:param name="parent" />
    <td style="width: 100%">
      <textarea style="width: 100%">
        <xsl:attribute name="name">
          param/<xsl:value-of select="normalize-space($parent)"/>
          <xsl:value-of select="../@name"/>
        </xsl:attribute>
      </textarea>
    </td>
  <td style="vertical-align: middle">
    <em>List of <xsl:value-of select="./@name"/></em>
  </td>
</xsl:template>
<xsl:template match="simple-type[@enum='yes']">
  <xsl:param name="parent" />
  <td style="width: 100%">
    <select style="width: 100%">
      <xsl:attribute name="name">
        param/<xsl:value-of select="normalize-space($parent)"/>
        <xsl:value-of select="../@name"/>
      </xsl:attribute>
      <xsl:for-each select="./enum-value">
        <option>
          <xsl:attribute name="value">
            <xsl:value-of select="."/>
          </xsl:attribute>
          <xsl:value-of select="."/>
        </option>
      </xsl:for-each>
    </select>
  </td>
  <td style="vertical-align: middle;">
    <em><xsl:value-of select="./@name"/></em>
```

```
    </td>
  </xsl:template>
  <xsl:template match="simple-type">
    <xsl:param name="parent" />
      <td style="width: 100%">
        <input style="width: 100%" >
          <xsl:attribute name="name">
            param/<xsl:value-of select="normalize-space($parent)"/>
            <xsl:value-of select="../@name"/>
          </xsl:attribute>
        </input>
      </td>
      <td style="vertical-align: middle;">
        <em><xsl:value-of select="./@name"/></em>
      </td>
  </xsl:template>
  <xsl:template match="complex-type">
    <xsl:param name="parent" />
    <td style="padding: 0;" colspan="2">
      <table style="width= 100%; border: thin solid">
        <xsl:apply-templates select="./*">
          <xsl:with-param name="parent" select="normalize-space($parent)" />
        </xsl:apply-templates>
      </table>
    </td>
  </xsl:template>
  <xsl:template match="element">
    <xsl:param name="parent" />
      <tr>
  <td style="vertical-align: middle">
        <strong><xsl:value-of select="./@name"/></strong>
      </td>
      <xsl:apply-templates select="./*">
        <xsl:with-param name="parent">
          <xsl:value-of select="$parent"/>
          <xsl:value-of select="../../@name"/>/
        </xsl:with-param>
      </xsl:apply-templates>
    </tr>
  </xsl:template>
</xsl:stylesheet>
```

Apêndice C

Listagem dos Serviços Testados

Todos os serviços listados neste apêndice estão disponíveis no repositório de serviços *X-Methods* (XMethods, 2004).

C.1 Serviços que a biblioteca WSDL não conseguiu processar

Real Time OFAC Check

Permite checagens ao escritório de controle de propriedades estrangeiras (EUA).

Romulan Numbers XLII

Converte entre numerais romanos e decimais.

WorldTime

Obtém o horário atual de um país a partir de seu código.

C.2 Serviços que necessitam de cadastro

Death Index

Retorna dados sobre o índice de mortes nos EUA.

ICQ Sender

Envia mensagens de texto para clientes de ICQ.

SmartPayments

Serviço de pagamento, com suporte a cartões de crédito e débito.

StrikeIron Econometric Statistics

Estatísticas sobre a economia norte-americana providas pelo *Federal Reserve Bank*.

StrikeIron Historical Stock Quotes

Informações históricas sobre preços de ações.

TAP Sender

Envia mensagens TAP.

XigniteReleases

Notícias divulgadas pelas principais fontes das seguradoras nos EUA.

C.3 Serviços com informações insuficientes

APR/Lease Webservice

Permite o cálculo de preços de aluguéis.

BGZip

Busca de endereços baseada em códigos postais búlgaros.

HCPCS

Contém os procedimentos e códigos alfanuméricos HCPCS de nível dois.

MailLocate

Encontra um endereço de *e-mail* atual baseado em um antigo.

StammService

Serviços para obter informações sobre um usuário da rede “OLI-it”.

SendSMSWorld

Envia mensagens SMS para diversos países.

C.4 Serviços não-visuais

SecureXML

Assina e verifica a assinatura de um documento XML.

Web RTF2HTML

Converte documentos RTF para HTML e vice-versa.

C.5 Serviços cuja invocação resultou em erro

Anagram

Serviço para a geração de anagramas. Durante a invocação o servidor não suportava o formato SOAP enviado.

Currency Exchange Rate

Calcula a taxa de câmbio entre duas moedas. O serviço era invocado mas não respondia (*timeout*).

C.6 Serviços que foram testados com sucesso

Capelo

Conversão de unidade de temperatura.

CountryWebservice

Permite obter: unidade monetária, código de discagem internacional, e código ISO de um país.

Currency Convertor

Taxa de conversão entre duas unidades monetárias.

Daily Dilbert

Retorna um URL ou um fluxo binário para uma tira de quadrinhos do personagem Dilbert, alterada diariamente.

DictService

Busca definições de palavras em inglês em diversos dicionários.

Dun & Bradstreet Business Verification

Verifica uma entidade de negócio (empresa) e sua localização.

eSynapsFeed

Artigos diários, dicas para programação .NET e exemplos.

GBNIR Holiday Dates

Permite a obtenção de datas de feriados móveis da Irlanda do Norte.

getJoke

Retorna uma piada aleatoriamente, dentro de uma categoria especificada.

Holiday Service

Permite a obtenção de datas de feriados de um dado país.

Host Info Web Service

Acessa servidores de DNS para obter informações de um *host* a partir de seu endereço IP ou nome.

Magic Squares

Retorna um Quadrado Mágico de um determinado tamanho.

Morse Code Translator WebService

Converte textos para código *Morse* e vice-versa.

NASCAR Winston Cup Statistics

Estatísticas atualizadas da copa NASCAR para o ano atual e anteriores.

NATTax

Retorna o estado, município e jurisdição especial de impostos a partir de informações de endereço do cliente.

Periodic Table

Retorna o peso atômico, símbolo e número atômico de qualquer elemento.

Quote of the Day

Retorna uma citação diferente a cada dia.

SendSMS

Envia mensagens SMS para a Índia.

SendEmail

Envia uma mensagem para um endereço de *e-mail*.

Shakespeare

Recebe uma frase de uma peça de William Shakespeare e retorna a fala, o orador e a peça associados.

StockQuotes Web Service

Provê valores de ações NASDAQ e LJSE baseados e múltiplos critérios.

StrikeIron Global Address Locator

Refina e corrige endereços internacionais.

StrikeIron Global Address Verification

Valida e melhora informações de contato para endereços globais.

TEXT to Braille

Converte textos para Braille e vice-versa.

US Weather

Oferece previsão de tempo para os próximos cinco dias a partir de um código postal norte-americano.

VideoGamesFinder

Informações sobre jogos eletrônicos.

Vigenere

Permite cifrar e decifrar textos.

Validate Email Address

Valida endereços de *e-mail* em seus servidores.

WorldCup Footballpool 2006

Resultados de jogos da copa do mundo FIFA 2006

XigniteExchanges

Informações sobre companhias de seguro ao redor do mundo.

Referências Bibliográficas

Abdelnur, A. e Hepper, S.; 2003. Java Portlet Specification. http://www.jcp.org/en/jsr/detail?id=168, acesso: Abril de 2006.

Adler, S., andJeff Caruso, A. B., Deach, S., Graham, T., Grosso, P., Gutentag, E., Milowski, A., Parnell, S., Richman, J., e Zilles, S.; 2001. Extensible Stylesheet Language. http://www.w3.org/TR/xslt/, acesso: Maio de 2006.

Agrawal, R., J.Bayardo Jr., R., Gruhl, D., e Papadimitriou, S.; 2005. Vinci: a service-oriented architecture for rapid development of Web applications. *Computer Networks*, 39(5):523–539. http://www.sciencedirect.com/science/article/B6VRG-45KNFVW-3/2/c95328f84e56d58736208744b7a72b3d, acesso: Novembro 2006.

Altheim, M. e McCarron, S.; 2001. XHTML 1.1 Specification. http://www.w3.org/TR/xhtml11/, acesso: Março de 2006.

Apache; 2000. Apache Struts. http://struts.apache.org/, acesso: Junho de 2006.

Apache; 2004. Xerces 2 Java Parser. http://xerces.apache.org/xerces2-j, acesso: Maio de 2006.

Apache; 2005. Axis (Apache eXtensible Interaction System). http://ws.apache.org/axis, acesso: Abril de 2006.

Apache; 2006. Jetspeed 2 Enterprise Portal. http://portals.apache.org/jetspeed-2/, acesso: Maio de 2006.

Ballinger, K., Ehnebuske, D., Ferris, C., Gudgin, M., Liu, C. K., Nottingham, M., e Yendluri, P.; 2004. Basic Profile Version 1.1. http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html, acesso: Março de 2006.

BEA Systems; 2004. Service-Oriented Architecture Solution Accelerator Guide. http://ftpna2.bea.com/pub/downloads/SOA_SAG.pdf, acesso : Abril de 2006.

Boyer, J. M., Landwehr, D., Merrick, R., Raman, T. V., Dubinko, M., e Leigh L. Klotz, J.; 2006. XForms 1.0. http://www.w3.org/TR/xforms/, acesso: Julho de 2006.

Bray, T., Hollander, D., e Layman, A.; 1999. Namespaces in XML. http://www.w3.org/TR/REC-xml-names/, acesso: Abril de 2006.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., e Yergeau, F.; 2004. Extensible Markup Language (XML) 1.0 (Third Edition). http://www.w3.org/Submission/xmlschema-api/, acesso: Abril de 2006.

Burns, E. e McClanahan, C. R.; 2004. JavaServer Faces Specification. http://www.jcp.org/en/jsr/detail?id=127, acesso: Junho de 2006.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., e Stal, M.; 1996. *Pattern-Oriented Software Architecture*. John Wiley & Sons, New York, NY, EUA.

Canfora, G., Corte, P., Nigro, A. D., Desideri, D., Penta, M. D., Esposito, R., Falanga, A., Renna, G., Scognamiglio, R., Torelli, F., Villani, M. L., e Zampognaro, P.; 2005. The C-Cube Framework: Developing Autonomic Applications Through Web Services. Em *Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, páginas 1–6, New York, NY, EUA. ACM Press.

Cerami, E.; 2002. *Web Services Essentials*. O'Reilly & Associates, Sebastopol, CA, EUA, 1a. edição.

Cesar, R.; 2006. O Lego do Software. *Exame*, 40(14).

Channabasavaiah, K., Holley, K., e Tuggle, E. M.; 2004. Migrating to a service-oriented architecture. http://www.cytetech.com/documents/SOA-IBM.pdf, acesso: Abril de 2006.

Chappell, D. A. e Jewell, T.; 2002. *Java Web Services*. O'Reilly & Associates, Sebastopol, CA, EUA, 1a. edição.

Chipara, O. e Slominski, A.; 2003. Xydra - An automatic form generator for Web Services. http://www.extreme.indiana.edu/xgws/xydra/, acesso: Maio de 2006.

Christensen, E., Curbera, F., e Weerawarana, G. M. S.; 2001. Web Services Description Language. http://www.w3.org/TR/wsdl, acesso: Abril de 2006.

Clark, J.; 1999. XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt/, acesso: Abril de 2006.

Clark, J. e DeRose, S.; 1999. XML Path Language. http://www.w3.org/TR/xpath/, acesso: Maio de 2006.

Clement, L., Hately, A., von Riegen, C., e Rogers, T.; 2004. UDDI Version 3.0.2. http://uddi.org/pubs/uddi_v3.htm, acesso: Maio de 2006.

Coward, D. e Yoshida, Y.; 2003. Java Servlet 2.4 Specification. www.jcp.org/en/jsr/detail?id=154, acesso: Abril de 2006.

Curbera, F., Ehnebuske, D., e Rogers, D.; 2002. Using WSDL in a UDDI Registry, Version 1.07. http://www.uddi.org/pubs/wsdlbestpractices-V1.07-Open-20020521.pdf, acesso: Maio de 2006.

da Silva, E. L. e Menezes, E. M.; 2005. *Metodologia da Pesquisa e Elaboração de Dissertação*. UFSC, Florianópolis, 4a. edição.

Duftler, M. J., Fremantle, P., e Kaputin, J.; 2005. Java APIs for WSDL 1.1. http://www.jcp.org/en/jsr/detail?id=110, acesso: Maio de 2006.

Eclipse; 2005. Eclipse Platform 3.1. http://www.eclipse.org/, acesso: Maio de 2006.

ECOLEAD; 2004. ECOLEAD - European Collaborative Networked Organisations Leadership Initiative. http://www.ecolead.org, acesso: Abril de 2006.

Fiorini, M.; 2006. *Uma Arquitetura Genérica de Software para Disponibilização de uma Aplicação Web para Dispositivos Móveis*. Dissertação (mestrado em engenharia elétrica), Centro Tecnológico, Universidade Federal de Santa Catarina, Florianópolis.

Gannon, D., Alameda, J., Chipara, O., Christie, M., Dukle, V., Fang, L., Farrellee, M., Kandaswamy, G., Kodeboyina, D., Krishnan, S., Moad, C., Pierce, M., Plale, B., Rossi, A., Simmhan, Y., Sarangi, A., Slominski, A., Shirasuna, S., e Thomas, T.; 2005. Building grid portal applications from a web service component architecture. *Proceedings of the IEEE*, 93(3):551–563.

GNU; 1999. GNU Lesser General Public License. http://www.gnu.org/licenses/lgpl.html, acesso: Abril de 2006.

Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., e Nielsen, H. F.; 2003. SOAP Version 1.2. http://www.w3.org/TR/soap/, acesso: Abril de 2006.

Haas, H. e Brown, A.; 2004. Web Services Glossary. http://www.w3.org/TR/ws-gloss/, acesso: Janeiro de 2006.

Hendricks, M., Galbraith, B., Irani, R., Miberny, J., Modi, T., Tost, A., amd S. Jeelani Basha, A. T., e Cable, S.; 2002. *Profissional Java Web Services*. Alta Books, 1a. edição.

Hepper, S.; 2004. Comparing the JSR 168 Java Portlet Specification with the IBM Portlet API. http://www-128.ibm.com/developerworks/websphere/library/techarticles/0406_hepper/0406_hepper.html, acesso: Março de 2006.

Hepper, S., Fischer, P., Hesmer, S., Jacob, R., e Taylor, D. S.; 2005. *Portlets and Apache Portals*. Manning Publications, Greenwich, CT, EUA, 1a. edição.

Huang, Y. e Chung, J.-Y.; 2003. A web services-based framework for business integration solutions. *Electronic Commerce Research and Applications*, 2(1):15–26. http://www.sciencedirect.com/science/article/B6X4K-48642HS-1/2/a751ffc1be1676f5b9955ea9050c160d, acesso: Junho 2006.

Hunter, J. e Crawford, W.; 2001. *Java Servlet Programming*. O'Reilly & Associates, Sebastopol, CA, EUA.

Husted, T., Dumoulin, C., Franciscus, G., e Winterfeldt, D.; 2003. *Struts in Action*. Manning Publications, Greenwich, CT, EUA, 1a. edição.

IBM; 2006a. WebSphere Portal. http://www.ibm.com/software/genservers/portal/, acesso: Julho de 2006.

IBM; 2006b. WebSphere Portlet Factory. http://www.ibm.com/software/genservers/portletfactory/, acesso: Julho de 2006.

Jardim-Gonçalves, R., Grilo, A., e Steiger-Garção, A.; 2006. Challenging the interoperability between computers in industry with MDA and SOA. *Computers in Industry*, 57(8-9):679–689. http://www.sciencedirect.com/science/article/B6V2D-4K9C542-1/2/18e0e027bb5e97d7c36b9f190f8cba08, acesso: Julho de 2006.

Java; 2004. Java Standard Edition 5.0. http://java.sun.com/j2se/1.5.0/, acesso: Maio de 2006.

Jayanti, K. e Hadley, M.; 2006. SOAP with Attachments API for Java 1.3. http://www.jcp.org/en/jsr/detail?id=67, acesso: Abril de 2006.

Johnson, R. E. e Foote, B.; 1988. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35. http://www.laputan.org/drc/drc.html, acesso: Junho de 2006.

Kelly, K. E., Kratky, J. J., Speicher, S., e Wells, K.; 2006. XML Forms Generator. http://www.alphaworks.ibm.com/tech/xfg?open&ca=drs-aw-ecl&S_TACT=106AH21W&S_CMP=AWRSSECL, acesso: Maio de 2006.

Krasner, G. E. e Pope, S. T.; 1988. A cookbook for using the model-view controller user interface paradigm. *Journal of Object-Oriented Programming*, 1(3):26–49.

Kropp, A., Leue, C., e Thompson, R.; 2003. Web Services for Remote Portlets Specification. http://www.oasis-open.org/committees/wsrp, acesso: Março de 2006.

Litani, E.; 2004. XML Schema API. http://www.w3.org/Submission/xmlschema-api/, acesso: Abril de 2006.

MacKenzie, C. M., Laskey, K., McCabe, F., Brown, P. F., e Metz, R.; 2006. Reference Model for Service Oriented Architecture 1.0. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm, acesso: Novembro de 2006.

Mann, K. D.; 2004. *JavaServer Faces in Action*. Manning Publications, Greenwich, CT, EUA.

Mindreef; 2006. Mindreef SOAPscope. http://www.mindreef.com/products/soapscope/index.php, acesso: Maio de 2006.

Musciano, C. e Kennedy, B.; 2002. *HTML and XHTML: The Definitive Guide*. O'Reilly & Associates, Sebastopol, CA, EUA, 5a. edição.

OASIS; 2004. UDDI Technical White Paper. http://uddi.org/pubs/uddi-tech-wp.pdf, acesso: Maio de 2006.

OMG; 2002. OMG IDL Syntax and Semantics. http://www.omg.org/cgi-bin/doc?formal/02-06-39, acesso: Janeiro de 2006.

OMG; 2004. Common Object Request Broker Architecture: Core Specification. http://www.omg.org/technology/documents/formal/corba_2.htm, acesso: Julho de 2006.

Open Knowledge Initiative; 2006. Open Knowledge Initiative - Accelerated interoperability through simplified integration. http://www.okiproject.org/, acesso: Janeiro de 2006.

Peltz, C.; 2003. Web Services Orchestration. http://devresource.hp.com/drc/technical_white_papers/WSOrch/WSOrchestration.pdf, acesso: Outubro de 2006.

Polgar, J., Bram, R. M., e Polgar, A.; 2006. *Building and Managing Enterprise-Wide Portals*. Idea Group, Hershey, 1a. edição.

Rails; 2004. Ruby on Rails. http://www.rubyonrails.org/, acesso: Junho de 2006.

Raman, T. V.; 2003. *XForms: XML Powered Web Forms*. Addison-Wesley Longman Publishing, Boston, MA, EUA.

Roth, M. e Pelegrí-Llopart, E.; 2003. JavaServer Pages 2.0 Specification. http://www.jcp.org/en/jsr/detail?id=152, acesso: Abril de 2006.

Schach, S. R.; 2004. *Object-Oriented and Classical Software Engineering*. McGraw-Hill Professional.

Schaeckn, T.; 2002. Web Services for Remote Portals (WSRP) Whitepaper. http://www.oasis-open.org/committees/wsrp/documents/wsrp_wp_09_22_2002.pdf, acesso: Maio de 2006.

Singh, M. P. e Huhns, M. N.; 2005. *Service-Oriented Computing: Semantics, Processes, Agents*. John Wiley & Sons, New York, NY, EUA, 1a. edição.

SourceForge.net; 2006. Project Statistics for Dynamic SOAP Portlet. http://sourceforge.net/project/stats/detail.php?group_id=162231&ugn=soap-portlet&mode=alltime&type=prdownload, acesso: Julho de 2006.

SQLData System; 2003. Generic SOAP Client. http://soapclient.com/soaptest.html, acesso: Maio de 2006.

Telang, R. e Mukhopadhyay, T.; 2005. Drivers of Web portal use. *Electronic Commerce Research and Applications*, 4(1):49–65. http://www.sciencedirect.com/science/article/B6X4K-4DN978D-5/2/4cb1d3af0fb482f280d0fa9c41263395, acesso: Outubro 2006.

Thomas, D. e Hansson, D. H.; 2005. *Agile Web Development with Rails : A Pragmatic Guide*. Pragmatic Bookshelf, 1a. edição.

Thompson, H. S., Beech, D., Maloney, M., e Mendelsohn, N.; 2004. XML Schema (2nd Edition). http://www.w3.org/TR/xmlschema-1/, acesso: Abril de 2006.

Tramontin Jr., R. J.; 2004. *Configuração e Integração de Dados em Plataformas para Empresas Virtuais*. Dissertação (mestrado em engenharia elétrica), Centro Tecnológico, Universidade Federal de Santa Catarina, Florianópolis.

UDDI4J; 2005. UDDI4J. http://uddi4j.sourceforge.net, acesso: Maio de 2006.

Vassiliadis, B., Stefani, A., Tsaknakis, J., e Tsakalidis, A.; 2006. From Application Service Provision to Service-Oriented Computing: A Study of the IT Outsourcing Evolution. *Telematics and Informatics*, 23(4):271–293. http://www.sciencedirect.com/science/article/B6V1H-4H8830Y-1/2/85c98101943629ba52f55ceae9199dfd, acesso: Julho 2006.

Wang, H., Huang, J. Z., Qu, Y., e Xie, J.; 2005. Web services: problems and future directions. *Journal of Web Semantics*, 1(3):241–320.

WSDL4J; 2005. Web Services Description Language for Java Toolkit. http://sourceforge.net/projects/wsdl4j, acesso: Maio de 2006.

XMethods; 2004. XMethods. http://www.xmethods.net/, acesso: Julho de 2006.

XML Portlet; 2006. Xmlportlet, a Portlet using XML/XSLT to render mode views. https://xmlportlet.dev.java.net/, acesso: Maio de 2006.

Zhu, X. e Zheng, X.; 2005. A Template-Based Approach for Mass Customization of Service-Oriented eBusiness Applications. Em *Proceedings of the 7th international conference on Electronic commerce*, páginas 706–710, New York, NY, EUA. ACM Press.