UNIVERSIDADE FUMEC
FACULDADE DE CIÊNCIAS EMPRESARIAIS – FACE

ISMAEL DE SOUZA OLIVEIRA JÚNIOR

COMPARAÇÃO ENTRE *FRAMEWORKS* JAVA PARA DESENVOLVIMENTO DE *WEB SERVICES*: Axis2 e CXF

Belo Horizonte
2013

ISMAEL DE SOUZA OLIVEIRA JÚNIOR

COMPARAÇÃO ENTRE *FRAMEWORKS* JAVA PARA DESENVOLVIMENTO DE *WEB SERVICES*: Axis2 e CXF

Monografia apresentada à Universidade FUMEC, no curso de Ciência da Computação, apresentado à disciplina Trabalho de Conclusão de Curso.

Orientador: Prof. Flávio Velloso Laper
Co-Orientador: Prof. Ricardo Terra
Orientador ABNT: Prof. Osvaldo Manoel Corrêa

Belo Horizonte
2013

AGRADECIMENTOS

À Deus, primeiramente, por sempre estar comigo e dar sabedoria e paciência necessárias para chegar até aqui.

Aos meus pais, por me darem todo o apoio necessário para que o sonho de me tornar bacharel em Ciência da Computação se tornasse realidade.

À Lucy, por todo o amor e paciência demonstrados durante este estudo.

Ao professor Ricardo Terra, por todo o ensinamento passado e pela paciência e atenção demonstradas na orientação deste trabalho.

RESUMO

A necessidade de troca de informações entre empresas com diferentes ambientes (sistemas operacionais, linguagens de programação, etc.) fazem com que a utilização de *Web services* se apresente como a solução mais apropriada. No entanto, como se trata de uma tecnologia independente de linguagem de programação, existem implementações para as mais variadas linguagens de programação. Mais especificamente, a linguagem Java possui diversos *frameworks* que auxiliam no desenvolvimento de *Web services*. Diante disso, este estudo apresenta dois *frameworks* largamente utilizados para a linguagem Java: Axis2 e CXF. Mais importante, este estudo compara esses dois *frameworks* frente a diversos aspectos, tais como padrões adotados e complexidade de desenvolvimento. Como resultado, pôde-se observar que o *framework* CXF é mais indicado para ser adotado por fábricas de software principalmente por utilizar padrões já consolidados na comunidade Java e por possuir uma complexidade menor de desenvolvimento.

Palavras-chave: Arquitetura Orientada a Serviços, Serviços *Web*, Java, Axis2, CXF.

ABSTRACT

Information exchange is the basis for the communication among companies. However, companies usually rely on different operating systems, programming languages, etc. This scenario indicates the use of Web services (WS) as the most suitable solution. Although WS is a technology-independent solution, there are implementations for a wide range of programming languages. Specifically, there are several Java-based frameworks that assist developers to build WS. In view of such circumstances, this study describes two widely used Java-based frameworks: Axis2 and CXF. As a practical contribution, this study also compares these frameworks w.r.t. standards, complexity, etc. As a result, we could observe that CXF overcomes Axis2 because the development of CXF-based WS is more intuitive and relies on patterns already consolidated by the Java community.

Keywords: Service Oriented Architecture, Web services, Java, Axis2, CXF.

LISTA DE FIGURAS

Figura 1 – Fluxo de um *Web service*... 11
Figura 2 – Componentes da arquitetura do Axis2.. 23
Figura 3 – Hierarquias de descrição e contexto do Axis2... 24
Figura 4 – Sistema motivador.. 28
Figura 5 – Estrutura de diretórios do aplicativo Axis2 para *containers web*............... 29
Figura 6 – Componentes da arquitetura do Apache CXF... 36
Figura 7 – Estrutura do projeto `webservicecxf` no Eclipse.................................... 40
Figura 8 – Fluxo de uma mensagem no servidor CXF... 42
Figura 9 – Fluxo de uma mensagem no consumidor CXF.. 44

LISTA DE SIGLAS

ADB	Axis2 *Data Binding*
API	*Application Programming Interface*
AXIOM	Axis2 *Object Model*
CORBA	*Common Object Request Broker Architecture*
DOM	*Document Object Model*
DTD	*Document Type Definition*
EAI	*Enterprise Application Integration*
ESB	*Enterprise Service Bus*
HTML	*Hyper Text Markup Language*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*HyperText Transfer Protocol Secure*
IDE	*Integrated Development Environment*
IDL	*Interface Description Language*
IIOP	*Internet Inter-Orb Protocol*
JAXB	*Java Architecture for XML Binding*
JAX-RS	*Java API for RESTful Web Services*
JAX-WS	*Java API for XML-Based Web Services*
JBI	*Java Business Integration*
JCP	*Java Community Process*
JMS	*Java Message Service*
JSR	*Java Specification Request*
MEP	*Message Exchange Pattern*
POJO	*Plain Old Java Object*
OMG	*Object Management Group*
RPC	*Remote Procedure Call*
REST	*Representational State Transfer*
SMTP	*Simple Mail Transfer Protocol*
SCA	*Service-Component Architecture*
SEI	*Service Endpoint Interface*
SOA	*Service-Oriented Architecture*
SOAP	*Simple Object Access Protocol*
SDO	*Service Data Objects*

StAX	*Streaming API for XML*
TCP	*Transmission Control Protocol*
UDDI	*Universal Description, Discovery and Integration*
URI	*Uniform Resource Identifier*
URL	*Uniform Resource Locator*
WSDL	*Web Services Description Language*
WS-BPEL	*Web Service Business Process Execution Language*
W3C	*World Wide Web Consortium*
XML	*Extensible Markup Language*
XMPP	*Extensible Messaging and Presence Protocol*

Sumário

Introdução 9
1. WEB SERVICES..11
1.1. *Web services* baseados em XML...13
1.1.1. WSDL 13
1.1.2. UDDI...15
1.1.3. SOAP...15
1.2. *Web services* baseados em REST..17
1.3. Especificações para *Web services* em Java 17
1.3.1. JAX-WS 18
1.3.2. JAX-RS..19
1.4. Ambiente de desenvolvimento 19
1.5. Considerações finais do capítulo 19
2. APACHE AXIS2 21
2.1. Visão geral 21
2.2. Arquitetura..22
2.2.1. Information processing Model...23
2.2.2. XML *processing Model* 25
2.2.3. SOAP *processing Model*..25
2.2.4. Deployment Model...26
2.2.5. *Client* API...26
2.2.6. Transports 27
2.2.7. Code Generation 27
2.2.8. Data binding 27
2.3. Sistema Motivador...27
2.4. Implementação do serviço 28
2.5. Implementação do consumidor 31
2.6. Considerações finais do capítulo 33
3. APACHE CXF 34
3.1. Visão geral 34
3.2. Arquitetura..36
3.2.1. Bus 36
3.2.2. Front-end...37
3.2.3. Messaging & Interceptors...37

3.2.4. Data binding 37
3.2.5. Service model 38
3.2.6. Protocol binding 38
3.2.7. Transport 38
3.2.8. Modelo de processamento XML 39
3.3. Implementação do Serviço 39
3.4. Implementação do consumidor 42
3.5. Considerações finais do capítulo 44
Conclusão 46
REFERÊNCIAS 51

APÊNDICE I – Implementação do sistema motivador 54
APÊNDICE II – Implementação Axis2 56
APÊNDICE III – Implementação CXF 60

Introdução

A necessidade de troca de informações entre empresas com diferentes ambientes (sistemas operacionais, linguagens de programação, etc.) tem se tornado cada vez mais evidente. “Empresas podem ser compostas por várias aplicações que são customizadas, adquiridas a partir de terceiros, parte de um sistema legado[1], ou uma combinação disso, operando em diferentes plataformas de sistemas operacionais” (HOHPE, WOOLF, 2003, p. 31 – Tradução nossa). A partir dessa necessidade, surgiram os primeiros conceitos de integração de sistemas utilizando *Enterprise Application Integration* (EAI), que consiste no “processo de integração de vários sistemas de software que foram desenvolvidos de forma independente, usando tecnologias incompatíveis que continuam a ser gerenciados de forma independente” (RICHADSON, 2013 – Tradução nossa).

Entre as primeiras soluções de EAI podemos destacar duas implementações baseadas em *Remote Procedure Call (*RPC): CORBA e XML-RPC. RPC consiste de uma técnica para computação distribuída em sistemas baseados em cliente-servidor podendo ser considerada uma extensão da tradicional chamada local de procedimentos, de forma que o procedimento não precisa estar no mesmo ambiente da aplicação que deseja executá-lo (MARSHALL, 2013). CORBA é a sigla para *Common Object Request Broker Architecture*, uma arquitetura aberta, definida pela OMG[2] e independente de fornecedor e infraestrutura, que as aplicações usam para trabalhar em conjunto através da rede. Apesar de ser uma solução independente de linguagem de programação, possui como desvantagem necessitar que todos os sistemas de software envolvidos sejam baseados em CORBA (CORBA FAQ, 2013). Já o XML-RPC permite utilizar os recurso de RPC através da Internet sobre o protocolo HTTP[3]. No entanto, não recebeu apoio e caiu em desuso.

Atualmente, a solução mais utilizada para integração de sistemas se baseia na *Service-Oriented Architecture* (SOA), ou Arquitetura Orientada a Serviços, que é um conceito de arquitetura corporativo que promove a integração entre as áreas de negócio e Tecnologia da Informação (TI) por meio de um conjunto de interfaces de

---

[1] Segundo Sommerville (2003), sistemas legados são sistemas antigos que são fundamentais para as empresas que o utilizam e que sofreram (e ainda sofrem) várias alterações ao longo do tempo.
[2] *Object Management Group* (OMG) é um consórcio internacional que provê padrões para a área da Ciência da Computação (OMG, 2013).
[3] HTTP é definido como “o protocolo padrão usado para transportar informações entre servidores e clientes na Internet” (TANENBAUM; WETHERALL, 2011).

serviços acoplados. No entanto, não é correto afirmar que SOA é um modelo de integração de sistemas, nem mesmo que integração de sistemas contém SOA. “Podemos determinar que SOA é uma estratégia que tem por objetivo criar todos os ativos de software[4] de uma empresa via metodologia de programação orientada a serviços” (KOCH, 2013). De um ponto de vista mais técnico, notou-se que *Web service* possui as características necessárias para ser considerado a abordagem mais apropriada para implementar sistemas de software baseados em SOA.

Web service é um componente de software que pode ser implementado por diversas linguagens de programação e tem a capacidade de reutilizar componentes de aplicação e fazer com que sistemas desenvolvidos em diferentes linguagens de programação se comuniquem sem requerer que os sistemas envolvidos passem por larga reestruturações. Mais importante, pelo fato de ser implementado por diversas linguagens de programação, atualmente existem uma diversidade de *frameworks*[5] para as várias linguagens disponíveis no mercado. Diante desse cenário, o objetivo desse trabalho é descrever dois *frameworks* feitos para linguagem Java – Apache Axis2 e Apache CXF – e, por meio de uma análise comparativa de aspectos relevantes – tais como complexidade e padrões suportados – determinar qual *framework* é o mais apropriado para ser adotado por fábricas de software. Esse estudo, portanto, poderá apoiar pequenas e médias empresas de desenvolvimento de software na escolha de qual *framework* adotar no desenvolvimento de seus projetos.

Esta monografia está organizada como a seguir. O Capítulo 1 aborda *Web services*, suas principais características e os padrões adotados pela linguagem Java. O Capítulo 2 descreve o projeto e implementação de *Web services* e consumidores usando o *framework* Apache Axis2. Como alternativa ao Axis2, o Capítulo 3 descreve o *framework* Apache CXF. Por fim, na Conclusão, é apresentada uma análise comparativa no intuito de determinar qual *framework* é o mais indicado a ser adotado por fábricas de software.

---

[4] Vasconcellos (2013), define ativo de software como “todos os ativos que formam o grande inventário da área de TI, aquele que parece mais esquecido ou, relegado a um segundo plano”.
[5] Fayad e Schmidt (1997) definem *framework* como “um conjunto de classes que colaboram para realizar uma responsabilidade para um domínio de um subsistema da aplicação”.

1. WEB SERVICES

Web services são componentes de software, independentes de plataforma para implementação de serviços e oferecem um alto grau de reúso e interoperabilidade. Basicamente, *Web services* são concebidos com o intuito de consumir e/ou disponibilizar serviços. Para consumir serviços, um aplicativo efetua uma requisição a um *Web service* e o seu retorno é tratado por esse *Web service*. Por outro lado, ao disponibilizar um serviço, o *Web service* é apenas invocado quando solicitado. No entanto, *Web services* podem ser constituídos por vários serviços, cada um possuindo uma finalidade, fazendo com que esses sejam capazes de consumir e disponibilizar serviços simultaneamente. A Figura 1 ilustra o fluxo de trabalho básico de um *Web service*.

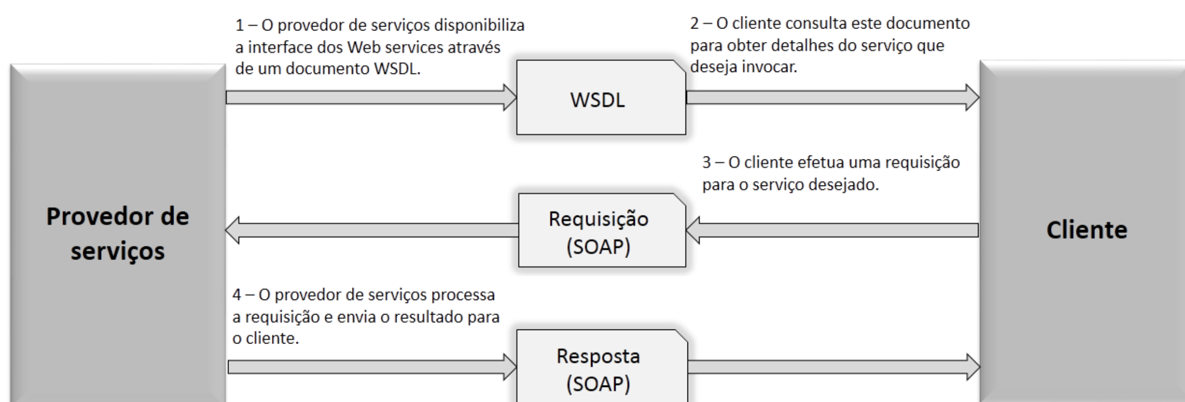

Figura 1: Fluxo de um *Web service*.
Fonte: JAVAWORLD, 2013. (Adaptado pelo autor)

Mais precisamente, a Figura 1 ilustra um provedor de serviços que publica um documento que descreve os serviços disponíveis (WSDL) e um cliente que consulta esse documento para obter detalhes do serviço que deseja invocar. O cliente ao efetuar a requisição desse serviço para o provedor utiliza o protocolo SOAP, o provedor processa essa requisição e envia o resultado para o cliente (utilizando também SOAP).

As duas principais funções dos *Web services* são: reutilizar componentes de aplicação e conectar sistemas de software diferentes (W3SCHOOLS, 2013). Essas funções fazem com que *Web service* seja a principal tecnologia utilizada para implementar sistemas de software cuja arquitetura é baseada em SOA. No entanto, é

necessário entender o que é um serviço e qual o seu papel dentro de um contexto SOA. Erl (2009, p. 25) define da seguinte forma:

> Os serviços existem como programas de software fisicamente independentes, com características de design distintas que dão suporte à obtenção dos objetivos estratégicos associados à computação orientada a serviços. Cada serviço recebe seu próprio contexto funcional distinto e possui um conjunto de capacidades relacionadas a esse contexto. Essas capacidades adequadas para a invocação por programas externos são comumente expressas via um contrato de serviços público.

Outra característica importante dos *Web services* é a sua capacidade de se compor com outros serviços. Essa característica recebe o nome de *Composição* e permite que um novo serviço seja criado com um nível ainda maior de abstração. Segundo Erl (2009, p. 228), *Web services* estabelecem um nível de composição nativo, fazendo com que esse seja mais um motivo para reforçar que *Web service* é a abordagem mais conveniente para construir SOA. A implementação de Composição de serviços mais conhecida é a *Orquestração*. Essa implementação consiste em um *Web service* que invoca outros serviços respeitando uma ordem de execução que é gerenciada por um controlador central. A tecnologia mais utilizada para implementar Orquestração é a *Web Service Business Process Execution Language* (WS-BPEL).

Existem ainda outras duas implementações de composição de serviços: *Coordenação* e *Coreografia*. *Coordenação* é um tipo de composição muito parecido com a *Orquestração*. No entanto, não existe uma ordem para execução dos serviços que participam desse tipo de composição.[6] Já a Coreografia descreve o fluxo entre as entidades envolvidas nesse tipo de composição, porém, o controle e o cumprimento desse fluxo é de responsabilidade de cada entidade.[7]

Ao longo deste capítulo serão descritos dois tipos de *Web services*: baseados em XML[8] (que é o foco deste trabalho) e baseados em REST. Em seguida, serão abordadas as especificações de desenvolvimento de *Web services* para a linguagem Java – JAX-WS e JAX-RS – e suas respectivas implementações de referência. Por último, é detalhado o ambiente de desenvolvimento utilizado ao longo desta monografia.

---

[6] O padrão que define *frameworks* para Coordenação é um conjunto de especificações: *WS-BusinessActivity* (WS-BA), *WS-AtomicTransaction* (WS-AT) e *WS-Coordination* (WS-C).

[7] Já o padrão para a implementação de Coreografia é definida pelo W3C e denominado por *Web Service Choreography Description Language* (WS-CDL).

[8] XML é a sigla para *Extensible Markup Language*. "É uma tecnologia para criar linguagens de marcação que descrevem dados de praticamente qualquer tipo de forma estruturada" (DEITEL, 2003).

1.1. *Web services* baseados em XML

Esse tipo de *Web service* utiliza XML para codificar e decodificar dados e utiliza um protocolo de transporte (geralmente HTTP) para trafegar esses dados. Atualmente denominado "*Web service* clássico", essa plataforma possui três elementos básicos: WSDL, UDDI e SOAP.

1.1.1. WSDL

Web Services Description Language (WSDL) é uma linguagem padronizada pelo W3C[9], baseada em XML, que é usada para descrever e localizar *Web services*. Assim, para descrever um *Web service* é necessário existir um documento WSDL, conforme exemplificado a seguir:

```
1.  <message name="getTermRequest">
2.    <part name="term" type="xs:string"/>
3.  </message>

4.  <message name="getTermResponse">
5.    <part name="value" type="xs:string"/>
6.  </message>

7.  <portType name="glossaryTerms">
8.    <operation name="getTerm">
9.      <input message="getTermRequest"/>
10.     <output message="getTermResponse"/>
11.   </operation>
12. </portType>

13. <binding type="glossaryTerms" name="b1">
14.   <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http" />
15.   <operation>
16.     <soap:operation soapAction="http://example.com/getTerm"/>
17.     <input><soap:body use="literal"/></input>
18.     <output><soap:body use="literal"/></output>
19.   </operation>
20. </binding>
```

O exemplo acima ilustra o documento WSDL de um *Web service* de consulta de termos em um glossário. Nesse documento existem dois elementos `<message>` (linhas 1-3 e 4-6) que recebem os nomes de `getTermRequest` que faz a requisição de um termo e `getTermResponse` que faz o retorno referente ao termo requisitado. O elemento `<message>` define o nome das mensagens que poderão ser

[9] *World Wide Web Consortium* (W3C) é um consórcio internacional responsável por desenvolver padrões *Web* (W3C, 2013 – Tradução nossa).

utilizadas no *Web service* e os atributos e tipos de dados suportados por uma mensagem.

Outro elemento presente nesse exemplo é o `<portType>` (linhas 7-12) que é o elemento mais importante de um documento WSDL, pois nele que de fato é definido um *Web service*, isto é, as operações e as mensagens envolvidas. No exemplo, o elemento `<portType>` recebe o nome de `glossaryTerms` e a única operação suportada recebe o nome de `getTerm` (que busca um termo). Essa operação define uma mensagem para receber uma requisição (`input message`) e outra para enviar a resposta (`output message`). Esse tipo de operação em que é definida uma mensagem para requisição e uma para resposta é conhecido como *request-response*. Existem ainda outros três tipos de operações suportadas pelo elemento `<portType>`: *one-way*, *solicit-response* e *notification*. Operações do tipo *one-way* podem apenas receber uma mensagem, mas não retornam uma resposta. Já as operações do tipo *solicit-response* são capazes de enviar uma requisição e aguardar uma resposta. E, por último, as operações do tipo *notification* são capazes de enviar uma mensagem, porém, não aguardam resposta. É importante mencionar que os tipos de operação não são explicitamente descritos no documento WSDL.

O elemento `<binding>` (linhas 13-20) define o formato da mensagem e o protocolo utilizado para um *Web service*. O protocolo mais utilizado para *Web services* é o protocolo SOAP (detalhado na seção 1.1.3). Esse elemento possui dois atributos: `name` e `type`. O atributo `name` define o nome da ligação e o atributo `type` aponta para o elemento `<portType>` do documento. O elemento `<soap:binding>` (linha 14) possui dois atributos: `style` e `transport`. Nesse caso, o atributo `style` recebeu o valor `document`, mas também aceita `rpc`. Esse atributo indica como a ligação entre o documento WSDL e uma mensagem SOAP é traduzida. Já o atributo `transport` define qual o protocolo de transporte utilizado para transmitir uma mensagem SOAP e, nesse caso, foi utilizado HTTP. O elemento `<operation>` (linhas 15-19) define as operações expostas pelo elemento `<portType>`. Cada operação deve possuir uma ação correspondente que o SOAP deve realizar. Deve-se também especificar como a entrada e a saída de dados serão codificados. Nesse caso, foi utilizado `literal` (no qual as mensagens se referem a uma definição de *schema* concreto), porém, é possível utilizar `encoded` (nesse caso as mensagens referem-se a uma definição de

schema abstrata e uma mensagem concreta pode ser produzida aplicando uma codificação específica).

Em *Web services* baseados em XML, a concretização do conceito de interoperabilidade só é possível graças ao documento WSDL. Por se tratar de um documento XML, pode ser interpretado por todas as linguagens de programação que implementam *Web services*.

1.1.2. UDDI

Universal Description, Discovery and Integration (UDDI) é um serviço de diretório onde empresas podem registrar e procurar *Web services* baseados em XML. UDDI é um *framework* independente de plataforma construído na plataforma Microsoft .NET, utiliza documento WSDL para descrever as interfaces dos *Web services* e se comunica através do protocolo SOAP (detalhado na seção seguinte).

Antes da existência de UDDI, não havia uma forma padronizada para que uma empresa chegasse até seus clientes e fornecedores com informações sobre seus produtos e serviços através da Internet. Mais importante, também não era possível integrar sistemas e processos entre empresas diferentes.

1.1.3. SOAP

Simple Object Access Protocol (SOAP) é um protocolo de comunicação baseado em XML utilizado para estabelecer comunicação entre aplicações através do protocolo HTTP. Como é utilizado para acessar um *Web service*, SOAP é independente de linguagem e plataforma. Além disso, é recomendado pelo W3C. Uma mensagem SOAP é um documento XML que contém os elementos `<soap:Envelope>`, `<soap:Head>`, `<soap:Body>` e `<soap:Fault>`. Basicamente, o elemento `<soap:Envelope>` identifica o documento XML como uma mensagem SOAP, `<soap:Head>` contém informações de cabeçalho da mensagem, `<soap:Body>` armazena o conteúdo da mensagem transmitida e `<soap:Fault>` armazena as informações de *status* e erros. As regras sintáticas para uma mensagem SOAP são:

1. Obrigatoriamente ser codificada utilizando XML;

2. Utilizar *namespaces*[10] padrão SOAP *Envelope* e SOAP *Encoding* e
3. Não conter referências DTD[11] e instruções de processamento XML[12].

A seguir, é apresentado um exemplo de mensagem SOAP.

```
1.  POST /InStock HTTP/1.1
2.  Host: www.example.org
3.  Content-Type: application/soap+xml; charset=utf-8
4.  Content-Length: nnn

5.  <?xml version="1.0"?>
6.  <soap:Envelope
7.  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
8.  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

9.  <soap:Body xmlns:m="http://www.example.org/stock">
10.   <m:GetStockPrice>
11.     <m:StockName>IBM</m:StockName>
12.   </m:GetStockPrice>
13. </soap:Body>

14. </soap:Envelope>
```

As linhas 1-4 referem-se ao cabeçalho de uma requisição de conexão com o protocolo HTTP. Após estabelecer a conexão, o cliente pode enviar a mensagem SOAP utilizando HTTP.

O elemento `<soap:Envelope>` (linhas 6-14) é obrigatório e utiliza dois *namespaces*. Um se refere diretamente a esse elemento e o padrão utilizado é definido por uma URI[13]: `http://www.w3.org/2001/12/soap-envelope`. O segundo *namespace* corresponde a codificação SOAP e dos tipos de dados. A URI padrão correspondente a esse *namespace* é: `http://www.w3.org/2001/12/soap-encoding`.

O elemento `<soap:Body>` (linhas 9-13) é um elemento obrigatório em uma mensagem SOAP. Ele armazena o conteúdo da mensagem transmitida ao último *endpoint*[14]. Deve ser o último *endpoint*, pois dependendo da arquitetura, uma

---

[10] *Namespace* é utilizado para evitar colisões de nomes de *tags* em um documento XML (DEITEL, 2003. p. 164-165).
[11] *Document Type Definition* (DTD) "define a estrutura de um documento (ou seja, quais elementos, atributos, etc. são permitidos)" (DEITEL, 2003. p. 176). No entanto, foi substituído pelo XML Schema.
[12] Instruções de processamento XML são marcações especiais que permitem instruções específicas para a aplicação que está processando um documento XML (PROCESSING INSTRUCTIONS, 2013).
[13] Segundo Deitel (2003, p. 165-166), *Uniform Resource Identifier* (URI) "é uma série de caracteres usados para diferenciar nomes".
[14] *Endpoint* é uma associação entre um *Web service* e um endereço de rede, que comunica com uma instância de serviço, indicando o local, protocolo e tipos de dados (ENDPOINT REFERENCE, 2013).

mensagem SOAP pode trafegar por vários *endpoints*. Cada elemento filho de `<soap:Body>` pode ter seu próprio *namespace*.

1.2. *Web services* baseados em REST

Representational State Transfer (REST) é uma arquitetura definida para sistemas distribuídos que teve origem a partir de uma tese de doutorado escrita por Roy Fielding, um dos principais autores da especificação do protocolo HTTP. Segundo Fielding (2000), REST é um estilo híbrido derivado de várias arquiteturas baseadas em rede combinado com restrições adicionais que definem uma interface de conector uniforme.

Essa arquitetura está relacionada a recursos que são identificados por URI. Um recurso pode ser qualquer tipo de informação, por exemplo: pedido, cliente, fornecedor, etc. As informações referentes a esse recurso são as representações e são armazenadas em arquivos formatados, tais como HTML[15], XML ou JSON[16].

Web services podem ser construídos utilizando os princípios da arquitetura REST e são conhecidos como RESTful *Web services*. *Web services* desenvolvidos sobre essa abordagem são vistos como recursos e identificados por sua URI. Os *Web services* expõem um grupo de operações através de uma interface comum para transferência de estado entre clientes e recursos e essas operações utilizam métodos HTTP, tais como GET e POST. Os clientes especificam qual representação desejam declarando um dos métodos definidos nessa interface usando a URI sobre o protocolo HTTP (BALANI; HATHI, 2013). *Web services* baseados em REST são uma alternativa para os *Web services* tradicionais (baseados em XML). No entanto, a falta de padronização da arquitetura REST impede que essa seja amplamente empregada.

1.3. Especificações para *Web services* em Java

Como já mencionado, *Web services* são componentes de software padronizados pelo W3C e podem ser desenvolvidos em praticamente qualquer linguagem de programação. No entanto, cada linguagem possui sua própria especificação, ou implementação de referência a fim de auxiliar os desenvolvedores

---

[15]Segundo a HTML Introduction (2013), HTML é a sigla para *Hyper Text Markup Language* e é uma linguagem usada na criação de páginas *web.*

[16]JSON é um formato de texto independente de linguagem de programação baseado em um subconjunto da linguagem JavaScript utilizado para troca de dados. (JSON, 2013)

a manter o padrão. Para isso, a plataforma Java conta com a colaboração do *Java Community Process* (JCP). "JCP é o mecanismo para o desenvolvimento de especificações técnicas padrão da tecnologia Java" (JCP, 2013). O JCP utiliza os *Java Specification Requests* (JSRs) para descrever as especificações propostas e tecnologias que deseja adicionar na plataforma Java. Cada especificação é possui uma identificação única. Por exemplo, a especificação JAX-WS (detalhada a seguir) é descrita pela JSR 224. Mais especificamente, a linguagem Java possui uma especificação para *Web services* baseados em XML e outra para *Web services* baseados em REST: *Java API for XML-Based Web Services* (JAX-WS) e *Java API for RESTful Web Services* (JAX-RS).

1.3.1. JAX-WS

JAX-WS é uma especificação para desenvolvimento de *Web services* baseados em XML e de clientes para consumir serviços utilizando linguagem Java. Sua especificação é definida pelo JCP, identificada pela JSR 224 e atualmente encontra-se na versão 2.x. Originalmente, o nome dessa especificação era JAX-RPC. Na primeira versão da especificação, uma aplicação Java poderia ser vista como uma aplicação RPC, porém, em forma de *Web service*. JAX-WS manteve todas as características da versão anterior, mas aumentou significativamente o seu poder de expressão (KALIN, 2009). A especificação JAX-WS permite criar *Web services* a partir de código Java utilizando anotações[17] apropriadas. Em *Web services,* essa abordagem recebe o nome de *bottom-up*, pois o documento WSDL é gerado à partir do código-fonte. JAX-WS utiliza uma série de anotações que são derivadas de algumas especificações, por exemplo, JSR 181 que define a sintaxe de anotações para programação de *Web services* e JSR 250 que define anotações utilizadas na linguagem Java. Para fazer uma ligação entre os tipos de dados existentes em um documento XML e objetos Java, JAX-WS utiliza uma outra especificação: *Java Architecture for XML Binding* (JAXB). "JAXB é uma especificação que define como JavaBeans[18] podem ter dados ligados ao XML" (JAYASINGHE; AZEEZ, 2011, p. 180 – Tradução nossa).

[17] Anotações proveem informações sobre o código que está sendo escrito ou até mesmo do próprio programa. Em contraste com comentários, anotações são interpretadas por compiladores (TI EXPERT, 2013).

[18] JavaBeans, ou simplesmente *beans*, são componentes de software reutilizáveis que podem ser desenvolvidos para criar aplicações sofisticadas (JAVABEANS, 2013).

A implementação de referência para JAX-WS é desenvolvida como um projeto *open source*[19] e é parte do projeto GlassFish, um servidor de aplicação também *open source* para aplicações Java.

1.3.2. JAX-RS

JAX-RS é uma especificação para desenvolvimento de *Web services* baseados na arquitetura REST para a linguagem Java. É também uma especificação definida pelo JCP na JSR 339 e encontra-se na versão 2.x. Por fim, Jersey é a implementação de referência para essa especificação.

1.4. Ambiente de desenvolvimento

Como já definido, o objetivo desta monografia é comparar dois *frameworks* amplamente utilizados para desenvolvimento de *Web services* utilizando linguagem Java – Axis2 e CXF – que serão respectivamente apresentados nos capítulos 2 e 3. Como o foco é dado nos *Web services* baseados em XML, o estudo dos *frameworks* é regido pela especificação JAX-WS. O ambiente de desenvolvimento utilizado para programar os *Web services* e seus consumidores é descrito a seguir:

- **Java Development Kit (JDK) 7u17**: conjunto de ferramentas que permitem criar sistemas de software na plataforma Java.
- **Eclipse Juno (4.2)**: *Integrated Development Environment* (IDE) para desenvolvimento de programas.
- **Maven 3.0.4**: ferramenta utilizada para gerenciar projetos de desenvolvimento de sistemas de software, principalmente voltados à plataforma Java.
- **Apache Tomcat 7.0.35:** *container* de aplicações web construídas em linguagem Java utilizando a especificação Java EE (JSR 316).

1.5. Considerações finais do capítulo

Neste primeiro capítulo foram abordados conceitos relacionados a *Web services*, componentes de software implementados independentes de linguagem de programação utilizados para reutilizar componentes de software e/ou integrar

[19] Open-source, ou código aberto, é uma metodologia de desenvolvimento de software que tem como premissa a liberação do código-fonte anexo ao programa (OPENSOURCE, 2013 – Tradução nossa).

sistemas de software distintos. Por possuir tal característica, é considerado a tecnologia mais apropriada para SOA. Outra característica importante dos *Web services* é a Composição, que é capacidade de um *Web service* se compor com outros serviços. A implementação mais conhecida de Composição de serviços é a orquestração, que consiste em um *Web service* que invoca outros *Web services* respeitando uma ordem de execução.

Neste capítulo também foram demonstrados dois tipos de *Web services* e seus conceitos básicos: os baseados em XML (também conhecidos como *Web services* clássicos) e baseados em REST (conhecidos como RESTful *Web services*). Embora cada linguagem de programação possua sua própria implementação de *Web services*, este trabalho foca em implementações Java. Mais especificamente, o foco se dará nos *frameworks* Axis2 e CXF. Ainda mais importante, o estudo avalia apenas as implementações de *Web services* clássicos (baseados em XML) que seguem a especificação JAX-WS. O principal motivo de não avaliarmos RESTful *Web services* se dá pelo fato de não serem uma implementação padronizada e também por não serem tão empregados pelo mercado quanto os *Web services* clássicos.

No próximo capítulo é abordado o Axis2, um dos *frameworks* utilizado no desenvolvimento de *Web services* para a linguagem Java. Seu objetivo consiste basicamente em demonstrar a arquitetura do Axis2 e implementações de serviços e consumidores por meio do uso desse *framework*.

2. APACHE AXIS2

Apache Axis2 é um *framework open-source* utilizado no desenvolvimento de *Web services* para as linguagens Java e C. Na sua versão para Java, Axis2 suporta o desenvolvimento de *Web services* baseados em XML e baseados em REST.[20]

Na seção 2.1 é demonstrada uma visão geral desse *framework*, sua história, principais características e ferramentas. Na seção 2.2 é apresentada a arquitetura do Axis2. Na seção 2.3 é apresentado o sistema motivador utilizado pelos dois *frameworks* abordados nesta monografia que é a base para realizar a comparação entre eles. Na seção 2.4 é demonstrado como implementar um *Web service* utilizando Apache Axis2 e, por fim, na seção 2.5 é demonstrado como implementar um consumidor utilizando esse *framework*.

2.1. Visão geral

O Apache Axis2 foi lançado em 2006 sendo considerado a terceira geração de *frameworks* para desenvolvimento de *Web services* da Apache. Seus antecessores são Apache SOAP e Apache Axis 1.0. O Axis2 foi criado para atender os novos padrões de *Web services*, pois não era viável alterar a arquitetura do Axis 1.0 (JAYASINGHE; AZEEZ, 2011. p. 19).

Axis2 implementa vários padrões de *Web services*. Essas implementações podem ser nativas, como o padrão *WS-Addressing* ou implementadas a partir de módulos. Os módulos são *plugins* que podem ser estendidos pelo Axis2 e que implementam padrões de *Web service*, tais como: Apache Sandecha2 que implementa o padrão *WS-ReliableMessaging*, Apache Kandula2 que implementa os padrões *WS-Coordination* e *WS-AtomicTransaction* e Apache Rampart que implementa o padrão *WS-Security*.

Entre as características principais do Axis2, destacam-se o seu próprio modelo de objeto – o Axis2 *Object Model* (AXIOM) – e a sua capacidade de implantar serviços em um servidor de aplicação em execução. Para a implantação em execução, basta inserir no diretório de serviços o arquivo com a extensão `.class` que representa um *Web service* e o próprio servidor se encarrega de realizar o *deploy* e disponibilizar o serviço. A arquitetura do Axis2 é bastante flexível e permite que o

---

[20] Não faz parte do escopo do trabalho abordar a implementação para a linguagem C desse *framework*, a qual é detalhada em: `http://axis.apache.org/axis2/c/core/index.html`.

desenvolvedor inclua extensões para customizar qualquer característica do *framework*. Por exemplo, Axis2 oferece suporte a *Web services* assíncronos e MEPs[21] e possui uma clara e simples abstração da camada de transporte, fazendo com que o núcleo do Axis2 seja totalmente independente de protocolo de transporte.

Axis2 também possui ferramentas de apoio para o desenvolvimento de serviços e/ou consumidores. *Code Generation Tool* faz parte do núcleo do Axis2 e é um *plugin* que cria classes Java a partir de um WSDL. Existem também outros *plugins* para gerenciadores de projeto, como o Maven, e para IDEs, como o Eclipse, conforme pode ser observados na Tabela 1.

TABELA 1 – Ferramentas do Apache Axis2

Tipo de Ferramenta	Nomes
Plugins para Maven	Maven2 WSDL2Code, Maven Java2WSDL, Maven2 MAR *Plugin*, Maven2 AAR *Plugin*
Plugins para IDE	Code Generator Wizard - IntelliJ IDEA Plug-in, Code Generator Wizard - Eclipse Plug-in, Service Archive Wizard - Eclipse Plug-in

Fonte: Apache Axis2 (2013).

2.2. Arquitetura

Conforme pode ser observado na Figura 2, o Axis2 possui uma arquitetura modular, o que permite que componentes presentes nesse tipo de arquitetura sofram modificações sem impactar o núcleo da aplicação. Quando o Axis2 foi projetado, três regras fundamentais foram incorporadas à sua arquitetura para garantir maior flexibilidade e tornar o processamento de mensagens SOAP mais robusto: (i) separar a lógica do estado da aplicação, pois *Web services* são aplicações *stateless*, ou seja, não armazenam o seu estado anterior; (ii) possuir um modelo de informações único, o que permite ao Axis2 suspender e retomar suas atividades; e (iii) capacidade de suportar novas especificações de *Web services* realizando o mínimo de alterações possíveis nos módulos principais da sua arquitetura.

[21] *Message Exchange Pattern* (MEP) "é um *template* que estabelece um padrão para troca de mensagens entre duas partes que se comunicam" (MEPS, 2013 – Tradução nossa).

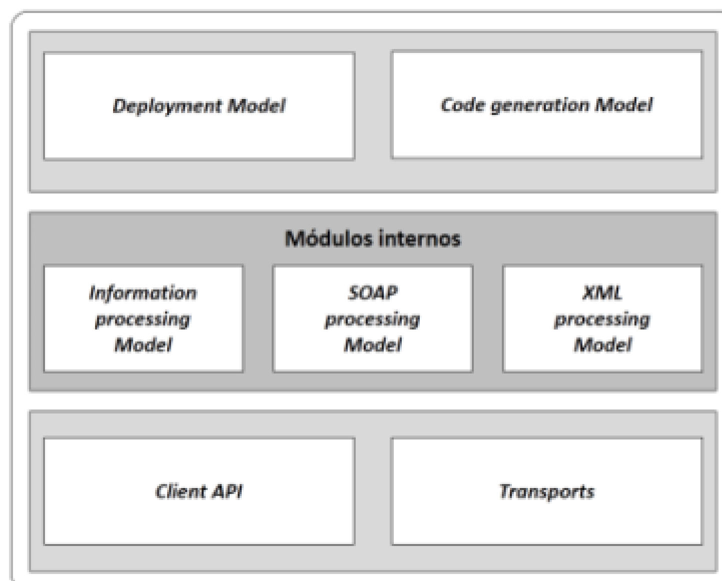

Figura 2: Componentes da arquitetura do Axis2.
Fonte: JAYSINGHE; AZEEZ, 2011, p.28. (Adaptado pelo autor)

Segundo Jayasinghe e Azeez (2011), a arquitetura do Axis2 é composta por componentes principais e secundários. A Figura 2 ilustra todos os componentes presentes na arquitetura do Axis2 das quais pode-se citar como principais: *Information processing Model*, *XML processing Model*, *SOAP processing Model*, *Deployment Model*, *Client API* e *Transports*. Já os seguintes módulos são considerados secundários: *Code Generation* e *Data binding*. Esses componentes são detalhados nas sub-seções seguintes.

2.2.1. Information processing Model

Esse componente define um modelo para manipulação de informações. Esse modelo consiste de duas hierarquias: (i) hierarquia de descrição e (ii) hierarquia de contexto. A hierarquia de descrição representa dados estáticos que podem ser, por exemplo, arquivos de configuração utilizados pelo Axis2. Já a hierarquia de contexto mantém informações dinâmicas sobre objetos que possuem mais de uma instância. Ambas as hierarquias criam um modelo que permite a busca de informações por meio de mapas chave-valor. Manter esses dados separados provê maior flexibilidade ao sistema, o que é fundamental para um *Web service*.

A hierarquia de descrição é muito importante para garantir outra característica fundamental para *Web services*: o desempenho. Dentro do conceito da hierarquia de descrição, o Axis2 possui uma outra hierarquia que armazena os dados

de configuração de uma forma mais organizada, que é a hierarquia de objetos. Segundo Jaysinghe e Azeez (2011), existem três tipos de arquivos de configuração, cada um em um nível, que populam e configuram essa hierarquia: (i) nível global (`axis2.xml`); (ii) nível de serviço (`services.xml`); e (iii) nível de módulo ou extensão de serviço (`module.xml`). O arquivo `axis2.xml` contém as configurações mínimas para iniciar o Axis2, seja como servidor ou consumidor e pode ser customizado de acordo com a necessidade. Já o arquivo `services.xml` armazena informações necessárias para implantar um *Web service*. Por fim, o arquivo `module.xml` armazena todas as configurações necessárias para criação de um módulo, que pode ser entendido como uma extensão de um serviço.

Já a hierarquia de contexto só é acionada quando o Axis2 recebe alguma mensagem. Como já definido, essa hierarquia armazena dados dinâmicos e em tempo de execução. Esses dados são compartilhados entre várias invocações ou entre os manipuladores de uma invocação. Nessa hierarquia, a principal forma de armazenamento de configurações é a utilização de `Properties`. A Figura 3 ilustra o relacionamento entre as hierarquias de descrição e contexto.

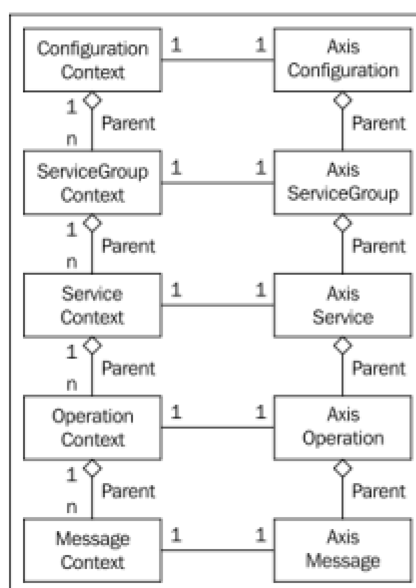

Figura 3: Hierarquias de descrição e contexto do Axis2.
Fonte: Jaysinghe e Azeez (2011).

De acordo com a Figura 3, para cada elemento da hierarquia de descrição deve existir um contexto associado. Como se trata de uma hierarquia, o elemento de

nível superior é responsável pelos elementos subsequentes, que podem não existir como podem ser vários.

2.2.2. XML *processing Model*

Como já citado na seção 2.1, o Axis2 possui seu próprio mecanismo para processamento de documentos XML, o AXIOM, que é uma implementação de StAX[22] e pode ser manipulado como qualquer modelo de objeto. O AXIOM surgiu para substituir o *Document Object Model* (DOM), que era utilizado como mecanismo de processamento de XML no Axis1. A principal desvantagem de utilização do DOM é que se faz necessário carregar a mensagem completa em memória antes de iniciar o processamento. No entanto, a principal vantagem da utilização do AXIOM é que os objetos são criados apenas quando necessário (instanciados sob demanda). Isso faz com que o consumo de memória seja menor, aumentando significativamente o desempenho.

2.2.3. SOAP *processing Model*

Esse componente é responsável por receber, enviar e processar mensagens SOAP. Uma característica da arquitetura do Axis2 é que o processamento de mensagens no núcleo do Axis2 é feito exclusivamente através do protocolo SOAP, ou seja, todas as mensagens devem ser convertidas para o formato SOAP antes de serem processadas. A arquitetura do Axis2 provê dois fluxos para realizar duas ações básicas em mensagens SOAP. Essas ações estão representadas na classe `AxisEngine` através dos métodos `send()` e `receive()`. Os dois fluxos recebem o nome de *Out Pipe* e *In Pipe* respectivamente e a combinação entre esses dois fluxos é dada através de MEPs. Esses fluxos trabalham em conjunto com os manipuladores de mensagens. Os manipuladores atuam como interceptadores sobre parte da mensagem SOAP que está sendo processada, provendo serviços adicionais. Os fluxos *Out Pipe* e *In Pipe* consistem de manipuladores de mensagens que tratam exclusivamente da saída e entrada de mensagens respectivamente.

---

[22] *Streaming API for XML* (StAX) é uma especificação para linguagem Java identificada pela JSR 173 que provê uma API padrão para manipulação de arquivos XML (JAYASINGHE; AZEEZ, 2011. p. 41).

2.2.4. Deployment Model

Esse componente provê um mecanismo concreto para configuração do Axis2 que consiste de três arquivos já definidos na seção 2.2.1: (i) `axis2.xml`; (ii) `services.xml`; e (iii) `module.xml`.

Jayasinghe e Azeez (2011) definem que o Axis2 provê um mecanismo de implantação semelhante àquele definido pela especificação Java EE, além do conceito de *deployer* que provê uma maneira de implantação de serviços simplificada. Axis2 oferece ainda suporte aos conceitos de *hot deployment* e *hot update*, isto é, respectivamente a capacidade de implantação e atualização sem a necessidade de parar o serviço.

2.2.5. *Client* API

Esse módulo consiste de uma API para criação de consumidores de *Web services* e possui duas classes: (i) `ServiceClient` e (ii) `OperationClient`. `ServiceClient` é utilizada apenas para enviar e receber documentos XML. Já `OperationClient` é utilizada para tarefas avançadas como trabalhar com cabeçalhos de mensagens SOAP.

A classe `ServiceClient` conta com o método `sendRobust()`, que possui a funcionalidade de enviar uma requisição para um *Web service*, mas não recebe um retorno (apenas uma exceção, caso ocorra algum erro). Apesar do Axis2 ser construído para suportar qualquer interação entre mensagens, nativamente suporta apenas dois MEPs: *One-way* e *In-Out*. *One-way* trata o fluxo da mensagem unidirecional (entrada ou saída), enquanto que o *In-Out* trata mensagens nos dois fluxos (entrada e saída). Os demais métodos da classe `ServiceClient` são baseados nesses dois MEPs. O método `fireAndForget()` é baseado no MEP *One-way* e apenas envia uma requisição e não se encarrega de receber uma mensagem de retorno, nem mesmo uma exceção. Os métodos `sendReceive()` e `sendReceiveNonBlocking()` são baseados no MEP *In-Out* e são utilizados para enviar requisições e aguardar o retorno do *Web service*. O método `sendReceive()` é o mais utilizado e o método `sendReceiveNonBlocking()` é utilizado apenas para chamadas assíncronas.

2.2.6. *Transports*

Axis2 possui duas construções básicas para transportes: *Transport Senders* e *Transport Receivers* e que podem ser definidos nas configurações globais do Axis2. É através do *Transport Receiver* que o `AxisEngine` recebe as mensagens, enquanto o *Transport Sender* é responsável por enviar mensagens. Mais importante, o núcleo do Axis2 é independente do tipo de transporte (*sender* ou *receiver*). Por fim, Axis2 é construído com suporte aos seguintes protocolos de transporte: HTTP, HTTPS, TCP, SMTP, JMS e XMPP.

2.2.7. *Code Generation*

Code Generation é uma ferramenta disponibilizada pelo Axis2 que tem o intuito de gerar código-fonte para *Web services* e consumidores a partir de documentos WSDL. Essa característica recebe o nome de *top-down*, o que facilita e agiliza o desenvolvimento de *Web services.* Algumas possibilidades de utilização dessa ferramenta se dá através de *plugins* para o Maven e para IDEs como o Eclipse.

2.2.8. *Data binding*

No Axis2 o módulo *Data binding* é o responsável por fazer o mapeamento entre objetos Java e tipos de dados XML. Na arquitetura do Axis2 esse módulo fica fora do seu núcleo. Isso permite que um serviço ou consumidor seja criado utilizando diferentes *frameworks* de *Data binding*, por exemplo: Axis2 *Data Binding* (ADB), XMLBeans, JibX e JAXB-RI.

2.3. Sistema Motivador

Com o intuito de demonstrar o funcionamento de um *Web service* utilizando os dois *frameworks* estudados nesta monografia, foi construído um sistema de consulta de partidas de futebol. Basicamente, esse sistema recebe os parâmetros necessários para efetuar uma consulta a uma partida de futebol e o retorno consistirá de dados relevantes da partida consultada, como pode ser observado na Figura 4.

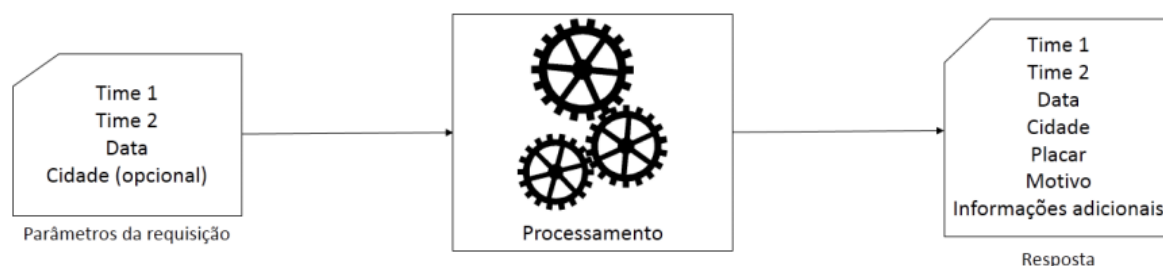

Figura 4: Sistema motivador

O sistema motivador recebe como parâmetros obrigatórios os nomes dos dois times envolvidos e a data do jogo (opcionalmente pode ser informado a cidade onde o jogo ocorreu). Após realizar uma consulta na base histórica e caso encontre alguma partida, é retornado um objeto `Partida`. Esse objeto contém os nomes dos times envolvidos, a data, cidade, placar e motivo da partida e as informações adicionais. A entidade `Partida` está descrita no Apêndice I, assim como a classe `EfetuaConsultaPartida` que realiza de fato a consulta e é de fato independente do *framework* de *Web service* utilizado.

2.4. Implementação do serviço

O Axis2 suporta a criação de três tipos serviços: SOAP, RESTful e CORBA. É importante lembrar mais uma vez que o foco desta monografia é nos *Web services* clássicos, ou seja, baseados em XML utilizando protocolo SOAP. Para execução dos *Web services* é necessário a instalação do aplicativo `axis2` para *containers* de aplicações *web*.[23] Um *Web service* desenvolvido em Axis2 deve sempre ser implantado nesse aplicativo. A Figura 5 ilustra a estrutura de diretório desse aplicativo utilizando o Tomcat como *container web*.

Como pode ser visualizado na Figura 5, o aplicativo `axis2` deve ser implantado no diretório `webapps`, que é o diretório onde todas as aplicações gerenciadas pelo Tomcat são armazenadas. O diretório `axis2-web` armazena os arquivos necessários para acessar o aplicativo através de um navegador *web*. O diretório `META-INF`, possui o arquivo `MANISFEST.MF`, necessário para o *deploy* deste aplicativo no Tomcat. O diretório `WEB-INF` armazena arquivos necessários para realização de *deploy* de *Web services*. Os *Web services* podem ser publicados nos

[23] `http://axis.apache.org/axis2/java/core/download.cgi`.

diretórios pojo ou services, de acordo com a abordagem utilizada para o seu desenvolvimento (detalhada logo a seguir).

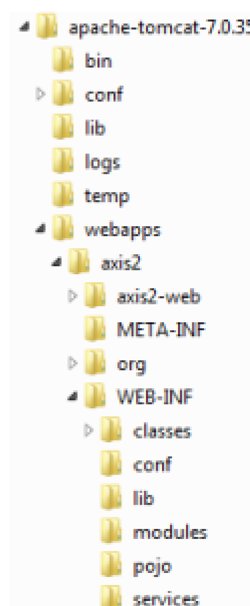

Figura 5: Estrutura de diretórios do aplicativo Axis2 para *containers web*.
Fonte: Do autor.

O Axis2 suporta quatro abordagens para o desenvolvimento de *Web services* clássicos: (i) utilização de AXIOM, que permite que o serviço seja criado utilizando a especificação StAX, baseando-se no objeto OMElement. Essa abordagem exige que o arquivo services.xml seja configurado de forma específica. Assim, para publicar um *Web service* utilizando essa abordagem é necessário criar um arquivo com a extensão .aar e o *deploy* deve ser realizado no diretório services. Para criação do arquivo .aar, pode-se utilizar o *plugin* axis2-aar-maven-plugin para o Maven; (ii) utilização do *WSDL2Java*, uma ferramenta que permite criar *Web services* através de um documento WSDL, e é possível, entre outras personalizações, especificar o componente *Data binding*. Para publicar um *Web service* através desse método também é necessário criar um arquivo .aar; (iii) utilização de POJO[24], que permite que a implementação do serviço seja feita em uma única classe e somente sua versão compilada é necessário para a implantação do serviço (isto é, o arquivo .class correspondente). Utilizando essa abordagem, o *deploy* deve ser realizado no diretório pojo do aplicativo Axis2. A utilização de POJO

[24] *Plain Old Java Object* (POJO) é o nome dado àquelas classes cuja regra de negócio é implementada em uma única classe simples (FOWLER, 2013).

pode não ser a mais apropriada para criação de *Web services*, pois, realizar o *deploy* de classes que utilizam pacotes, ou algum recurso da orientação a objetos é uma tarefa complexa; e (iv) utilização de anotações, que permite criar *Web services* baseando-se na especificação JSR 181 (detalhada a seguir).

Em relação a utilização de anotações (item iv), o Axis2 oferece duas formas principais para realizar o *deploy* de um *Web service*. A primeira consiste na criação de um POJO, na qual a classe que representa o serviço possui a implementação e as anotações necessárias. Porém, as desvantagens de utilizar POJO já foram mencionadas e as características do sistema utilizado nesta monografia não permitem que essa técnica seja utilizada. A segunda forma é semelhante à criação de um POJO, no entanto, utiliza-se o *plugin* `axis2-aar-maven-plugin` para criar um arquivo com a extensão `.aar`, permitindo que o *deploy* seja realizado no diretório `services` do aplicativo Axis2. Apesar da utilização de anotações, é necessário a existência do arquivo `services.xml` para realização do *deploy* do *Web service*. O código abaixo demonstra a classe que representa um *Web service* em Axis2. A implementação do sistema motivador – que é de fato a implementação do serviço – está detalhado no Apêndice I.

```
1.  package br.com.ismael;
2.  
3.  import java.util.GregorianCalendar;
4.  import javax.jws.WebMethod;
5.  import javax.jws.WebParam;
6.  import javax.jws.WebService;
7.  import br.com.ismael.EfetuaConsultaPartida;
8.  import br.com.ismael.Partida;
9.  
10. @WebService(serviceName = "ConsultaPartida")
11. public class ConsultaPartida {
12. 
13.   @WebMethod(operationName = "consultaPartida")
14.   public Partida consultaPartida(@WebParam(name = "nomeTime1") String
15.       nomeTime1, @WebParam(name = "nomeTime2") String nomeTime2,
16.       @WebParam(name = "data") GregorianCalendar data,
17.       @WebParam(name = "cidade") String cidade) {
18.     return new EfetuaConsultaPartida().consultaPartida(nomeTime1,
19.       nomeTime2, data, cidade);
20.   }
21. }
```

Como pode ser observado no código acima, a anotação `@WebService` (linha 10) identifica essa classe como um *Web service*. A anotação `@WebMethod` (linha 13) é opcional e é utilizada para customizar as operações de *Web service*. Essa

anotação provê o nome da operação e os elementos de ação que serão utilizados para customizar o atributo `name` do elemento `operation` e o elemento `soapAction` no documento WSDL. Outra anotação (embora opcional) é `@WebParam` (linhas 14, 15, 16 e 17) que é utilizada para customizar o nome dos atributos da requisição. Caso não seja utilizado, no documento WSDL são atribuídos nomes padrões para esses parâmetros.

2.5. Implementação do consumidor

Axis2 possui no seu núcleo uma API para criação de consumidores de *Web services* já descrita na seção 2.2.5. Entre as opções já apresentadas é demonstrada a implementação de um consumidor com Axis2 utilizando a classe `ServiceClient`. Os códigos abaixo representam apenas parte da implementação do consumidor, o código-fonte completo encontra-se disponível no Apêndice II.

```
1.  public OMElement getPartida(String nomeTime1, String nomeTime2,
2.        GregorianCalendar data, String cidade) {
3.
4.      OMFactory fac = OMAbstractFactory.getOMFactory();
5.      OMNamespace omNs = fac.createOMNamespace("http://ismael.com.br",
6.        "xs");
7.      OMElement operacao = fac.createOMElement("consultaPartida",
8.        omNs);
9.      OMElement time1 = fac.createOMElement("nomeTime1", omNs);
10.     time1.addChild(fac.createOMText(time1, nomeTime1));
11.     operacao.addChild(time1);
12.
13.     OMElement time2 = fac.createOMElement("nomeTime2", omNs);
14.     time2.addChild(fac.createOMText(time2, nomeTime2));
15.     operacao.addChild(time2);
16.
17.     OMElement dataPartida = fac.createOMElement("data", omNs);
18.     XMLGregorianCalendar xmlGregorianCalendar = null;
19.     try {
20.       xmlGregorianCalendar = DatatypeFactory.newInstance().
21.       newXMLGregorianCalendar(data);
22.     } catch (DatatypeConfigurationException e) {
23.       e.printStackTrace();
24.     }
25.     dataPartida.addChild(fac.createOMText(dataPartida,
26.       xmlGregorianCalendar.toString()));
27.     operacao.addChild(dataPartida);
28.
29.     OMElement cidadePartida = fac.createOMElement("cidade", omNs);
30.     cidadePartida.addChild(fac.createOMText(cidadePartida, cidade));
31.     operacao.addChild(cidadePartida);
32.
33.     return operacao;
34. }
```

Como já definido na seção 2.2.2, Axis2 utiliza AXIOM para processamento de XML, que é baseado na especificação StAX. O método `getPartida()` (linhas 1 e 2) recebe os parâmetros necessários para invocar o serviço `consultaPartida`. A classe `OMElement` é a base para processamento de XML utilizando StAX. Para instanciar um `OMElement` é necessário utilizar o objeto `OMFactory` (linha 4) e definir um *namespace*, por meio do objeto `OMNamespace` (linhas 5 e 6). A invocação do método `createOMElement()` (linhas 7, 9, 13, 17 e 29), como o próprio nome sugere, cria novos objetos `OMElement`. Ele recebe como parâmetro o nome de cada elemento a ser procurado no documento WSDL dentro de um *namespace* específico. Na linha 7 foi definido o elemento que representa a operação do *Web service* e os demais elementos representam parâmetros dessa operação. Cada parâmetro de operação é especificado pelo parâmetro correspondente do método `getPartida()` (linhas 10, 14, 25 e 30) e adicionado ao elemento que corresponde a operação (linhas 11, 15, 27 e 31). O retorno do método é o objeto `OMElement` que representa a operação. A seguir é demonstrado um trecho de código que utiliza o método descrito acima.

```
1.  EndpointReference endPoint = new EndpointReference(
2.        "http://localhost:8080/axis2/services/ConsultaPartida");
3.  ClienteAxis2 clienteAxis2 = new ClienteAxis2();
4.  String[] dataSeparada = fieldData.getText().split("/");
5.  XMLGregorianCalendar dataGregorian = DatatypeFactory.newInstance().
6.        newXMLGregorianCalendar();
7.  dataGregorian.setDay(new Integer(dataSeparada[0]));
8.  dataGregorian.setMonth(new Integer(dataSeparada[1]));
9.  dataGregorian.setYear(new Integer(dataSeparada[2]));
10. OMElement informacoesPartida = clienteAxis2.getPartida("Atético-MG",
11.       "Cruzeiro", dataGregorian, null);
12. Options options = new Options();
13. options.setTo(endPoint);
14. options.setTransportInProtocol(Constants.TRANSPORT_HTTP);
15.
16. ServiceClient cliente = new ServiceClient();
17. cliente.setOptions(options);
18. OMElement resposta = cliente.sendReceive(informacoesPartida);
```

O código acima representa a implementação de fato do consumidor de serviços. Nele, é indicado o *endpoint* que representa a URL do *Web service* (linha 1). O método `getPartida()` (linha 10) é invocado para realizar o tratamento necessário para invocar o serviço utilizando AXIOM. O objeto `Options` é utilizado para definir uma série de parâmetros que deseja adicionar a um `ServiceClient`. Nesse caso foram especificados a URL do *Web service* (linha 13) e o protocolo de transporte

utilizado (linha 14). Por fim, o objeto `ServiceClient` é instanciado (linha 16), recebe o objeto `options` (linha 17) e o método `sendReceive()` é invocado (linha 18) recebendo como parâmetro um objeto `OMElement` e retorna um outro objeto do mesmo tipo com o retorno do processamento do *Web service*.

2.6. Considerações finais do capítulo

Neste capítulo foi abordado o Apache Axis2 que é um *framework* largamente utilizado para auxiliar no desenvolvimento de *Web services* baseados em XML em REST para linguagem Java. Axis2 oferece suporte a vários padrões de *Web services*, protocolos de mensagem e transporte através de uma arquitetura modular. Essa arquitetura permite que alterações sejam feitas e recursos novos sejam adicionados ao Axis2 sem comprometer o seu núcleo, tornando essa arquitetura bastante flexível.

Axis2 também define um modelo para manipulação de informações que consiste em hierarquias de descrição e de contexto. A hierarquia de descrição representa os arquivos de configuração do Axis2 e a hierarquia de contexto mantém informações dinâmicas sobre objetos. Além disso, Axis2 possui o próprio processador de documentos XML – o AXIOM – que é baseado na especificação StAX. Outra característica importante é a utilização de um aplicativo próprio para *containers web* para realização de *deploy* dos *Web services*. No intuito de prover uma implementação de exemplo de serviços e consumidores, foi especificado um sistema simples de consulta de partidas de futebol.

Para implementar serviços, notou-se que existem quatro abordagens principais, sendo a abordagem por POJO menos apropriada, por limitações técnicas desse padrão e foi detalhado o desenvolvimento de *Web service* utilizando anotações definidas pela JSR 181. Por último, notou-se que o Axis2 possui uma API no seu núcleo para criação de consumidores de serviço, sendo detalhada a criação desses por meio da classe `ServiceClient`.

O próximo capítulo aborda o Apache CXF, outro *framework* largamente utilizado pelo mercado no desenvolvimento de *Web services* e consumidores para Java. O objetivo do capítulo 3 consiste em demonstrar a arquitetura desse *framework* e como são desenvolvidos *Web services* e consumidores.

3. APACHE CXF

Apache CXF é um *framework open-source* para a linguagem Java amplamente utilizado pelo mercado que provê suporte na criação e consumo de *Web services* utilizando as especificações JAX-WS e JAX-RS. Ainda oferece suporte a vários protocolos de mensagem e transporte. Balani e Hathi (2009, p. 20 – Tradução nossa) afirmam que o CXF "é desenvolvido com a missão de prover uma infraestrutura robusta para o desenvolvimento de *Web services* e facilitar o processo de desenvolvimento".

Na seção 3.1 é apresentado o surgimento desse *framework*, suas principais características e ferramentas. Na seção 3.2 é apresentada a arquitetura do CXF. Na seção 3.3 é demonstrado a implementação de um *Web service* utilizando Apache CXF. Na seção 3.4 é demonstrado como implementar um consumidor utilizando esse *framework* e, por fim, na seção 3.5 são feitas as considerações finais do capítulo.

3.1. Visão geral

Apache CXF surgiu a partir de dois projetos: Celtix e XFire e, por isso, o nome CXF. Celtix é um projeto *open-source* ESB[25] baseado na linguagem Java desenvolvido pela ObjectWeb, uma empresa que desenvolve soluções *open-source* de *middleware*. Já o XFire é um *framework open-source* baseado em Java para desenvolvimento de *Web services* baseados no protocolo SOAP desenvolvido pela Codehaus. Durante as versões iniciais de ambos os projetos, foi constatado que havia muitas características em comum entre eles e que era possível transformá-los em um único projeto. A partir dessa constatação, foi desenvolvido com a ajuda da Apache *Software Foundation*, o Apache CXF 2.0. Atualmente encontra-se na versão 2.7.4.

Apache CXF oferece suporte a vários padrões de *Web services* definidos por órgãos como W3C e OASIS, dos quais podemos destacar: SOAP, WS-I, WSDL, *WS-Addressing*, *WS-Discovery*, *WS-Policy*, *WS-ReliableMessaging*, *WS-Security*, *WS-SecurityPolicy*, *WS-SecureConverstation*, *WS-MetadataExcange* e parte do padrão *WS-Trust*. Sua arquitetura foi projetada para ser compatível com vários protocolos de mensagens (por exemplo SOAP, REST/HTTP e XML) e transporte (por exemplo, HTTP, JMS e TCP), diferentes formatos de arquivos (por exemplo,

[25] *Enterprise Service Bus* (ESB) é uma plataforma de integração que conecta camadas de serviços, permitindo que os consumidores tenham acesso a esses serviços (DIKMANS; LUTTIKHUIZEN, 2012).

documentos XML e JSON) e ferramentas de associações de dados entre tipos Java e XML (por exemplo, JAXB 2.x, Aegis, Apache XMLBeans, *Service Data Objects* (SDO) e JiBX). Utiliza APIs que permitem ligações adicionais ao CXF, fornecendo suporte a outros formatos de mensagens tais como CORBA/IIOP. Outra característica importante do CXF é a sua integração com o Spring *framework*.[26] Essa integração permite utilizar arquivos de configuração do Spring *framework* para que a publicação de *endpoints* seja feita de forma mais simples.

Apache CXF conta ainda com várias ferramentas de apoio para o desenvolvimento de serviços e/ou consumidores. Segundo Apache CXF (2013), existem ferramentas para geração de código, geração de documentos WSDL, adição de *endpoints*, geração de arquivos de suporte e validação de arquivos. A Tabela 2 lista tais ferramentas classificadas de acordo com o seu tipo. Como um exemplo, a ferramenta WSDL *Validation* auxilia o desenvolvedor na verificação de arquivos WSDL criados.

TABELA 2 – Ferramentas do Apache CXF

Tipo de Ferramenta	Nomes
Geração de código	*WSDL to Java*, *WSDL to JavaScript* e *Java to JavaScript*.
Geração de documento WSDL	*Java to WSDL*, *XSD to WSDL*, *IDL to WSDL* e *WSDL to XML*.
Adição de *endpoints*	*WSDL to SOAP*, *WSDL to CORBA* e *WSDL to service*.
Geração de arquivos de suporte	*WSDL to IDL*.
Validação de arquivos	*WSDL Validation*.

Fonte: Apache CXF (2013).

Web services desenvolvidos com CXF podem ser implantados em servidores web – Tomcat, por exemplo – ou em servidores de aplicação robustos que implementam a especificação Java EE, tais como Websphere, Weblogic, JBoss, Geronimo e JOnAS. Ainda, podem ser implementados em servidores baseados em SCA[27] como o Tuscany. CXF suporta também *Web services* implantados como *service engine* em *containers* JBI[28], como ServiceMix, OpenESB, e Petals.

---

[26] Spring *framework* é um *framework* de aplicações em camadas criado para simplificar o desenvolvimento de sistemas corporativos para a linguagem Java (BALANI; HATHI, 2009).
[27] *Service-component Architecture* (SCA) é um grupo de especificações para o desenvolvimento de aplicações baseadas em SOA (SERVICE-COMPONENT ARCHITECTURE, 2013).
[28] *Java Business Integration* (JBI) é uma especificação definida pela JSR 208 que define uma abordagem para implementar SOA utilizando WSDL (JAVA BUSINESS INTEGRATION, 2013).

3.2. Arquitetura

A arquitetura do CXF é baseada nos componentes *Bus*, *Front-end*, *Messaging & Interceptors*, *Data binding, Service Model*, *Protocol bindings* e *Transport*. A Figura 6 ilustra os componentes desta arquitetura.

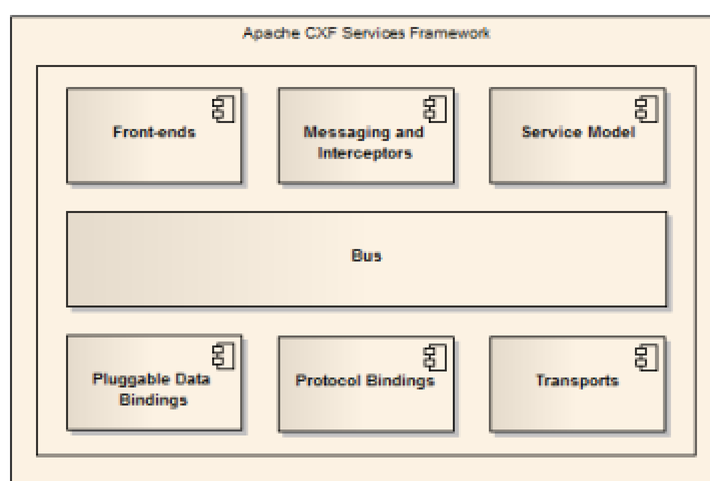

Figura 6: Componentes da arquitetura do Apache CXF.
Fonte: APACHE CXF (2013).

A Figura 6 mostra uma visão dos módulos arquiteturais do Apache CXF. É uma arquitetura bem flexível em que o componente *Bus* é um provedor de recursos e os demais componentes possuem responsabilidades específicas. As sub-seções seguintes detalham cada componente dessa arquitetura.

3.2.1. Bus

Bus é o componente principal da arquitetura do CXF. Ele é um provedor de recursos compartilhados para a execução do CXF dos quais podemos destacar gerenciadores de WSDL e *binding factory*. Esse componente pode ser configurado para a inclusão de recursos e/ou serviços próprios ou modificação dos recursos padrões. Isso é possível devido a técnicas de injeção de dependência.[29] A implementação padrão deste componente é baseada no Spring *framework*. A classe `SpringBusFactory` é responsável por criar o *Bus*. Essa classe procura arquivos de configuração localizados no diretório `META-INF/cxf` do projeto e cria os respectivos contextos de aplicação.

[29] Injeção de dependência é uma técnica que visa remover dependências desnecessárias entre as classes e ter um *design* de software que seja fácil de manter e evoluir (DEVMEDIA, 2013).

3.2.2. Front-end

Front-end provê um modelo de programação para interagir com o CXF e cada implementação é separada do restante do CXF. O CXF fornece as APIs de *front-end* JAX-WS, JAX-RS, *Simple* e JavaScript e elas funcionam por meio de interceptadores que são adicionados aos serviços e *endpoints*.

3.2.3. Messaging & Interceptors

Messaging é uma forma de estabelecer comunicação entre sistemas de software através de trocas de mensagens, o que garante baixo acoplamento entre as aplicações envolvidas (HAASE, 2013). O CXF possui uma camada de *Messaging* genérica composta por *Messages*, *Interceptors* e *Interceptor chains* que são representados pelas interfaces `Message`, `Interceptor` e `InterceptorChain`, respectivamente. O *Interceptor* é a unidade fundamental dessa camada e um dos elementos mais importantes da arquitetura do CXF. Seu objetivo é interceptar as mensagens trocadas entre clientes de *Web services* e componentes do servidor. No CXF isso é implementado através do conceito de *Interceptor chains* – representando pela classe `PhaseInterceptorChain` – que segundo Balani e Hathi (2009, p. 43) "é a funcionalidade principal da execução do CXF" e torna a arquitetura do CXF bastante flexível (APACHE CXF, 2013). A interceptação de mensagens pode ser reconfigurada em qualquer ponto do processamento, provendo ao CXF a capacidade de pausar e retomar as cadeias de interceptadores.

De um ponto de vista mais técnico, a interface `Interceptor` possui o método `handleMessage`, que permite atuar sobre as mensagens. Um ponto importante é que os *Interceptors* são unilaterais e desconhecem qual tipo de mensagem (requisição, resposta ou falha) estão lidando.

3.2.4. Data binding

Assim como no Axis2, *Data binding* é o responsável por fazer o mapeamento entre objetos Java e tipos de dados XML. Os tipos de componentes *data binding* que o CXF suporta são: JAXB, Aegis, Apache XMLBeans, SDO e JiBX (em desenvolvimento). Dentre eles, o JAXB é o padrão adotado.

3.2.5. Service model

Service model é a representação de um serviço dentro do CXF e é constituído de dois módulos: `ServiceInfo` e o próprio `Service`. `ServiceInfo` contém o modelo do serviço baseado no documento WSDL juntamente com suas operações, ligações, *endpoints* e *schema*. Já o `Service` contém o `ServiceInfo`, informações sobre *data binding*, interceptadores e propriedades do serviço, entre outras informações.

3.2.6. Protocol binding

Protocol binding é o componente que define o protocolo de comunicação utilizado pelos *Web services*. *Bindings* definem formas para mapear formatos concretos e protocolos sobre a camada de transporte. Na arquitetura do CXF, uma ligação contém duas classes principais: `BindingFactory` e `Binding`. `BindingFactory` instancia um objeto `Binding` que contém interceptadores específicos para realizar a ligação e também implementa o método `createMessage()` que cria uma implementação de `Message` específica para esse `Binding`. Atualmente, o CXF suporta os protocolos SOAP 1.1, SOAP 1.2, REST/HTTP, CORBA e XML nativo.

3.2.7. Transport

Esse componente define um protocolo de alto nível para transmissão de mensagens. Dentro desse conceito, existe o elemento *Conduit*, que provê a base para o envio de mensagens de saída. No CXF, a interface que representa o elemento *Conduit* possui o mesmo nome e a implementação desse elemento é representado pela classe `ConduitInitiator`. A interface `Conduit` possui dois métodos para tratamento de mensagens: (i) `prepare(message)`, que inicia o envio da mensagem; e (ii) `close(message)`, que fecha e descarta todos os recursos utilizados no envio da mensagem. Atualmente, o CXF suporta os protocolos HTTP, HTTPS, HTTP-Jetty, HTTP-OSGI, Servlet, local, JMS e In-VM, além dos protocolos SMTP/POP3, TCP e Jabber baseados no Apache Camel.

3.2.8. Modelo de processamento XML

Embora esse modelo não esteja ilustrado na Figura 6, é importante destacar que o CXF utiliza um padrão internacional para processamento de XML, o *Fast Infoset*. Trata-se de uma especificação que padroniza a codificação binária para *XML Information Set*.[30] Sua principal característica é utilizar técnicas para maximizar a velocidade de processamento de documentos XML.

3.3. Implementação do Serviço

O CXF suporta a implementação de três tipos principais de serviços: SOAP, RESTful e CORBA. Como já definido anteriormente, o foco deste trabalho é nos *Web services* "clássicos", ou seja, baseados em XML utilizando protocolo SOAP. Existem quatro abordagens para implementação de serviços utilizando o CXF: (i) `WSDL2Java`, que é um *plugin* que cria classes e interfaces Java de acordo com um documento WSDL; (ii) JAX-WS *Providers*, que permite que os serviços sejam criados a nível de mensagens, ao contrário da utilização do *plugin* e da abordagem tradicional (que é detalhada a seguir), que são criadas a nível de operação. Utilizando essa abordagem, existe apenas uma operação denominada `invoke` que recebe ou o conteúdo de uma mensagem SOAP (SOAP Body) ou uma mensagem completa (SOAP Envelope); (iii) módulo presente no CXF que permite a criação de serviços utilizando JavaScript através da biblioteca Java Rhino; e (iv) criar classes Java utilizando anotações apropriadas, que é a maneira mais tradicional de implementar serviços utilizando o CXF e é detalhada a seguir.

Na seção 1.4, em que foi descrito o ambiente de desenvolvimento utilizado nesta monografia, apontou-se o Maven como um dos elementos mais importantes para agilizar o desenvolvimento dos *Web services*. Maven é um gerenciador de construção de projetos e possui o conceito de *Archetype*. Segundo Maven (2013), *Archetype* é um conjunto de ferramentas para auxiliar na construção de projetos Maven. Existem *Archetypes* para vários tipos de projetos Maven, inclusive para criação de *Web services* utilizando o CXF.[31] Os passos para criação de um *Web service* utilizando Maven é descrito a seguir:

---

[30] *XML Information Set* é uma especificação que provê uma série de definições que devem ser utilizadas por outras especificações que necessitam obter informações de um documento XML (W3C, 2013).
[31] Não é objetivo deste estudo demonstrar como criar um ambiente Maven. Detalhes de configuração do ambiente Maven em: `http://maven.apache.org/run-maven/index.html`.

1. Em uma tela de terminal – sob o diretório do *workspace* – efetue uma busca nos repositórios Maven:
   `mvn archetype:generate -Dfilter=org.apache.cxf.archetype:`
2. Selecione a opção:
   `org.apache.cxf.archetype:cxf-jaxws-javafirst (Creates a project for developing a Web service starting from Java code).`
3. Selecione a última versão e defina a nomenclatura desejada.

Em seguida, basta importar o projeto no Eclipse. Ao importar o projeto, é possível visualizar que a base para a criação de um *Web service* utilizando CXF está pronta. A Figura 7 ilustra a estrutura do projeto no Eclipse sob a perspectiva Java EE.

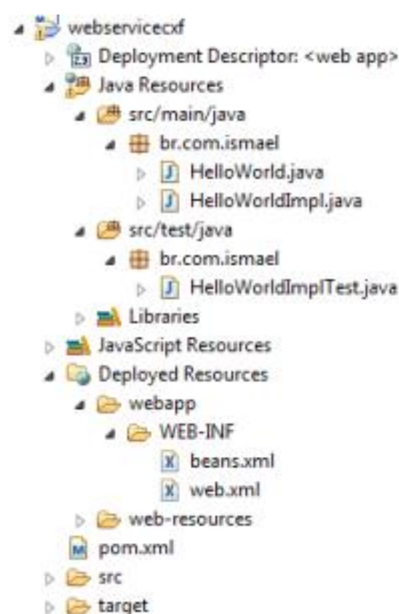

Figura 7: Estrutura do projeto `webservicecxf` no Eclipse.
Fonte: Do autor.

No diretório `main` foi criado um *Service Endpoint Interface* (SEI) chamado `HelloWorld.java`. Um SEI é definido como uma interface de *Web service* e é identificada através da anotação `@WebService` que é adicionada antes da declaração da interface. Essa interface deve expor os métodos que poderão ser invocados pelo consumidor. Foi criado também através do *Archetype* a implementação do serviço, que recebeu o nome de `HelloWorldImpl.java`. Essa classe deve implementar a

SEI, ou seja, representa o serviço. Essa classe também recebe a anotação `@WebService`, porém, é informado como parâmetro o nome completo da interface que representa o *endpoint*. Outro arquivo criado foi o `beans.xml`. Esse arquivo faz parte do Spring *framework* e facilita a declaração de *endpoints*. Para declarar um *endpoint* basta informar dentro da *tag* `<jaxws:endpoint>` um valor arbitrário e único no campo `id`, o nome da classe que implementa o serviço no campo `implementor` e o endereço utilizado para acessar o *Web service* que deve igual ao nome da SEI e declarado no campo `address`. Por último, foi criada uma classe de teste chamada `HelloWorldImplTest.java`. Essa classe representa um teste de unidade para o serviço criado.

É importante ressaltar que o projeto criado através de um *Archetype* facilita o desenvolvimento e o torna mais ágil, porém é apenas um exemplo. Isto é, para publicar o serviço desejado devem ser feitas alterações. O código abaixo demostra a SEI que publicará o serviço implementado pelo sistema motivador desta monografia. É importante notar que seu nome foi alterado para `ConsultaPartida.java`. A classe `ConsultaPartidaImpl.java` e o arquivo de configuração do Spring *framework* `beans.xml` estão detalhados no Apêndice III.

```
1.  package br.com.ismael;
2.  
3.  import java.util.GregorianCalendar;
4.  import javax.jws.WebMethod;
5.  import javax.jws.WebParam;
6.  import javax.jws.WebService;
7.  import javax.xml.bind.annotation.XmlElement;
8.  import br.com.ismael.Partida;
9.  
10. @WebService
11. public interface ConsultaPartida {
12. 
13.   @WebMethod
14.   Partida consultaPartida(@WebParam(name = "nomeTime1")
15.        @XmlElement(required = true) String nomeTime1,
16.        @WebParam(name = "nomeTime2") @XmlElement(required = true)
17.        String nomeTime2, @WebParam(name = "data")
18.        @XmlElement(required = true) GregorianCalendar data,
19.        @WebParam(name = "cidade") @XmlElement(required = false)
20.        String cidade);
21. }
```

A interface acima representa a descrição do serviço a ser disponibilizado. Essa SEI expõe apenas um método e esse recebe como parâmetro o nome de dois times, a data de realização da partida e o nome de uma cidade e retorna um objeto

`Partida` contendo as informações completas da partida. As anotações utilizadas para a implementação de um *Web service* com CXF são as mesmas utilizadas na implementação com Axis2. A diferença é a utilização da anotação `@XMLElement` (linhas 15, 16, 18 e 19), que faz parte do conjunto de anotações definidas pela especificação JAXB. Essa anotação permite customizar elementos XML e neste caso, utilizou-se para definir elementos obrigatórios e opcionais da requisição. A Figura 8 ilustra o fluxo de uma mensagem durante o processamento de uma requisição de acordo com a arquitetura do CXF.

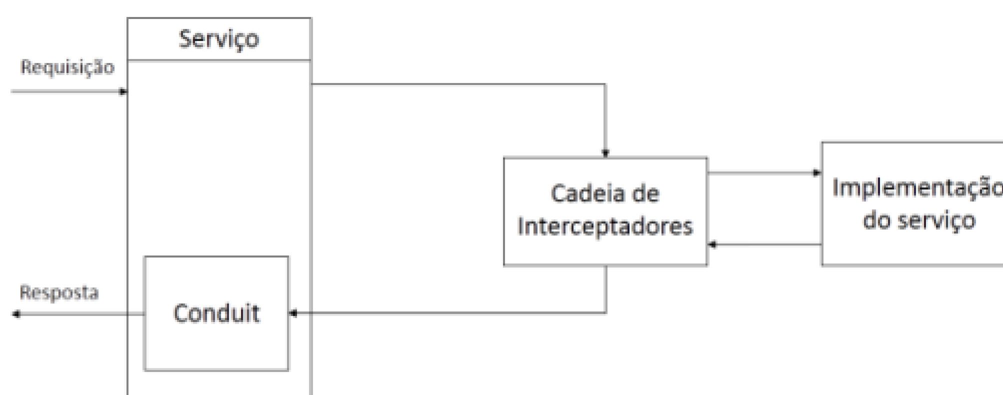

Figura 8: Fluxo de uma mensagem no servidor CXF.
Fonte: APACHE CXF, 2013. (Adaptado pelo autor)

A requisição inicial é interceptada pela cadeia de interceptadores que a manipula e a repassa para a implementação do serviço onde é de fato processada. Após essa etapa o resultado do processamento da requisição (resposta) é enviada para os *Interceptors chains* que realizam as manipulações e encaminha para o *Conduit*, onde a mensagem é preparada para ser enviada para o consumidor.

3.4. Implementação do consumidor

O CXF possui cinco diferentes abordagens para construção de consumidores de *Web services*: (i) `WSDL2java`, que é um *plugin* que cria classes Java com base em um documento WSDL. Existem três formas de utilização desse *plugin*: através da linha de comando, de um *plugin* Maven ou através da implementação da API `WSDL2Java`; (ii) JAX-WS *Proxy* que é um conceito presente na especificação JAX-WS. Utilizando essa abordagem, é necessário criar instâncias de serviços através da classe `Service` presente no pacote `javax.ws` e invocá-los: (iii) API

`Dispatch`, também presente no JAX-WS, a qual permite que serviços sejam invocados dinamicamente sem a necessidade de criar um cliente para isso. Basta criar uma mensagem, enviá-la para o servidor e aguardar o retorno: (iv) *front-end Simple*, em que os consumidores são instanciados baseados em serviços que são criados a partir desse tipo de serviço. O CXF possui a API `ClientProxyFactoryBean`. Essa API cria um cliente do tipo Java *Proxy*, permitindo utilizar uma interface para se comunicar com o *Web service*; e (v) *Dynamic clients*, a qual permite a criação de consumidores de *Web services* em tempo de execução. O código abaixo demonstra como criar um consumidor de *Web services* com CXF utilizando *Dynamic clients*. O código-fonte completo da classe abaixo é demonstrado no Apêndice III.

```
1.  DynamicClientFactory factory = DynamicClientFactory.newInstance();
2.  // Cria o cliente através do endereço do WSDL
3.  Client client = factory. createClient(
4.    "http://localhost:8080/webservicecxf/ConsultaPartida?wsdl");
5.  try {
6.    /*
7.     * Invoca o método através do nome e informa os parâmetros
8.     * necessários para a execução do mesmo
9.     */
10.   Object[] obj = client.invoke("consultaPartida", "Atlético-MG",
11.   "Cruzeiro", null);
12. } catch (Exception e1) {
13.   …
14. }
```

O CXF possui a classe `DynamicClientFactory` (linha 1) que cria clientes de *Web services* dinamicamente recebendo a URL do WSDL através do método `createClient` (linhas 3-4). Esse método retorna um objeto `Client` (que também faz parte do CXF). O objeto `Client` possui o método `invoke` (linha 10) que recebe o nome da operação e os parâmetros necessários para a execução do serviço. O retorno é um arranjo de `java.lang.Object` que armazena a resposta enviada pelo serviço e que pode ser manipulado da maneira necessária. A Figura 9 ilustra o fluxo de envio e recebimento de mensagens em um consumidor de *Web services* de acordo com a arquitetura do CXF.

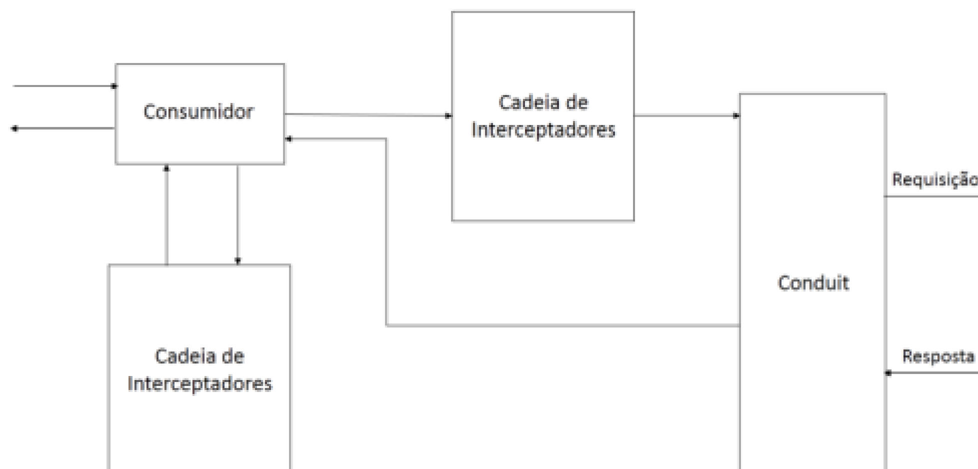

Figura 9: Fluxo de uma mensagem no consumidor CXF.
Fonte: APACHE CXF, 2013. (Adaptado pelo autor)

A Figura 9 mostra como é o processamento de uma mensagem para ser enviada (requisição) e para ser recebida (resposta). Para ser enviada, a requisição é criada pelo consumidor, repassada para o *Interceptor chain* que realiza o tratamento necessário na mensagem para que a mesma seja enviada para o *Web service* através do *Conduit*. Após o processamento do *Web service* a resposta é recebida pelo *Conduit* que a envia diretamente para o consumidor. No entanto, uma outro *Interceptor chain* pode interceptar essa mensagem, fazer algum tratamento e devolver a mensagem para o consumidor.

3.5. Considerações finais do capítulo

Neste capítulo foi abordado o Apache CXF. Trata-se de um *framework open-source* que surgiu a partir de dois projetos – Celtix e XFire. Em suma, é largamente utilizado para auxiliar no desenvolvimento de *Web services* para linguagem Java seguindo as especificações JAX-WS e JAX-RS e oferece suporte a vários padrões de *Web services*, protocolos de mensagem e transporte.

Observou-se que a arquitetura do CXF é bastante flexível, principalmente pelo módulo *Interceptor* que é responsável por interceptar e tratar mensagens entre clientes e serviços e tal interceptação pode ser customizada de acordo com a necessidade do desenvolvedor. Na implementação de serviços, observou-se que o CXF suporta várias formas para implementá-los, sendo a abordagem de criação de classes Java utilizando anotações específicas a mais empregada. A utilização do Maven torna o desenvolvimento de serviços significativamente mais ágil, pois o uso

de *Archetypes* gera um projeto de exemplo com classes e arquivos de configuração pré-definidos, bastando ao desenvolvedor customizá-los de acordo com o serviço que deseja implementar. Por fim, na implementação de consumidores observou-se que o CXF suporta também vários métodos e o método demonstrado – *Dynamic clients* – é simples e flexível, pois o seu retorno é um arranjo de objetos genéricos. Em outras palavras, qualquer objeto suportado pela linguagem Java.

Conclusão

A necessidade por troca de informações em ambientes heterogêneos, isto é, ambientes informatizados compostos por sistemas de softwares desenvolvidos em diferentes linguagens de programação, fazem com que a utilização de *Web services* se apresente como a solução mais apropriada. *Web service* consiste de componentes de software que podem ser implementados em qualquer linguagem de programação e tem por objetivo reutilizar componentes de software e/ou integrar sistemas de software distintos. Essa característica torna o *Web service* a tecnologia mais apropriada para implementar sistemas de software na arquitetura SOA. Outra característica importante dos *Web services* é a Composição, que é capacidade de um *Web service* se compor com outros serviços.

Por ser uma tecnologia largamente utilizada pelo mercado para integração de sistemas, cada linguagem de programação possui sua própria implementação de *Web services*, o que motivou a criação de diversos *frameworks* para auxiliar no seu desenvolvimento. Existem dois tipos de *Web services*: baseados em XML – também conhecidos como *Web services* clássicos – e baseados em REST – também conhecidos como RESTful *Web services*. No entanto, o foco desta monografia se deu na implementação de *Web services* baseados em XML utilizando a linguagem Java, mais especificamente em dois dos *frameworks* mais utilizados pelo mercado: Axis2 e CXF. Por outro lado, não foi abordado *Web services* baseados em REST devido à falta de padronização e por não serem largamente utilizados quanto os *Web services* clássicos.

Os capítulos 2 e 3 abordaram respectivamente os *frameworks* Apache Axis2 e Apache CXF. Como resultado da avaliação desses *frameworks*, a Tabela 3 provê uma análise comparativa através de doze aspectos considerados os mais relevantes. Baseando-se na Tabela 3, é definido a seguir uma comparação de cada aspecto citado entre os *frameworks* estudados.

1. **Flexibilidade para adicionar novos recursos**: O Axis2 possui uma arquitetura modular que o torna muito interessante, pois algumas implementações podem ser feitas fora do seu núcleo e incorporadas ao *framework*. Durante os estudos do CXF não foi identificado tal característica. Nesse aspecto, o Axis2 se mostrou superior ao CXF;

TABELA 3 – Análise comparativa entre os *frameworks* Axis2 e CXF

#	Característica	Axis2	CXF
1	Flexibilidade para adicionar novos recursos	Possui.	Não possui.
2	Padrões WS-* suportados	*WS-Addressing*, *WS-ReliableMessaging*, *WS-Coordination* e *WS-AtomicTransaction* e *WS-Security*.	WS-I, *WS-Addressing*, *WS-Discovery*, *WS-Policy*, *WS-ReliableMessaging*, *WS-Security*, *WS-SecurityPolicy*, *WS-SecureConverstation*, *WS-MetadataExcange* e parte do padrão *WS-Trust*.
3	Padrão para codificação de documentos XML	AXIOM.	*Fast Infoset*.
4	Protocolos de comunicação suportados	SOAP 1.1 e SOAP 1.2.	SOAP 1.1, SOAP 1.2, REST/HTTP, CORBA e XML nativo.
5	Protocolos de transporte suportados	HTTP, HTTPS, TCP, SMTP, JMS e XMPP.	HTTP, HTTPS, HTTP-Jetty, HTTP-OSGI, Servlet, local, JMS e In-VM, além dos protocolos SMTP/POP3, TCP e Jabber baseados no Apache Camel.
6	Padrões *Data binding* suportados	ADB, XMLBeans, JaxME, JibX, JAXB-RI.	JAXB, Aegis, XMLBeans, SDO e JiBX.
7	Criação de projetos através de *Archetypes* (Maven)	Apenas através de repositórios não-oficiais e não utilizam a especificação JAX-WS.	Suporta.
8	Ferramentas de apoio	Descritas na Tabela 1 (seção 2.1).	Descritas na Tabela 2 (seção 3.1).
9	Formas de *deploy* de *Web services*	O *deploy* deve ser feito sempre no seu próprio aplicativo para *containers web*, utilizando POJO ou arquivo com extensão própria (`.aar`).	O *deploy* pode ser feito um vários tipos de *containers web* e servidores de aplicação.
10	Formas de desenvolvimento de consumidores	Através de uma API própria presente na sua arquitetura.	Abordagens baseadas em JAX-WS.
11	Formas de depuração de erros	Através de testes de unidade através da própria IDE.	Permite que a depuração seja feita como qualquer outra aplicação, através de *deploy* na própria IDE.
12	Complexidade[32]	Apresenta alta complexidade para desenvolvimento de serviços e consumidores, devido ao uso de muitos arquivos de configuração e tecnologias proprietárias para processamento de documentos XML e *deploy* de serviços.	Apresenta baixa complexidade, pois possui uma arquitetura fácil de ser interpretada e busca implementar apenas especificações definidas pela comunidade Java e já consolidadas.

[32] A complexidade entre os dois *frameworks* foi analisada a partir de um único programador que possui três anos de experiência com a linguagem Java e é o autor desta monografia (Nota do autor).

2. **Padrões WS-* suportados**: Apesar de não implementar os padrões *WS-Coordination* e *WS-AtomicTransaction*, o CXF é mais completo, isto é, conta com uma quantidade maior de especificações implementadas em relação ao Axis2. Por isso, nesse aspecto o CXF é superior ao Axis2;

3. **Padrão para codificação de documentos XML**: O Axis2 conta com seu próprio mecanismo para processamento de documentos XML – o AXIOM – que possui como característica principal garantir alto desempenho. Por outro lado, o CXF utiliza um padrão internacional para essa tarefa: o *Fast Infoset*. Sua característica principal também é garantir um alto desempenho, porém, utilizando outras técnicas. Por utilizar um padrão internacional, o CXF é superior ao Axis2 também nesse aspecto;

4. **Protocolos de comunicação suportados**: O Axis2 suporta diferentes tipos de formatos de mensagens, mas, implementa apenas o protocolo de comunicação SOAP 1.1 e 1.2. Internamente, todas as mensagens são convertidas para o protocolo SOAP, o que é uma limitação da arquitetura. Já o CXF suporta, além dos mesmos protocolos do Axis2, os protocolos REST/HTTP, CORBA e XML nativo, fazendo com que nesse aspecto também o CXF seja superior ao Axis2;

5. **Protocolos de transporte suportados**: Nesse aspecto o CXF se mostra superior ao Axis2, pois implementa um número maior de protocolos de transporte;

6. **Padrões *Data binding* suportados**: Nesse aspecto, ambos os *frameworks* implementam praticamente os mesmos padrões de *Data binding*, com algumas exceções. A principal diferença está no padrão que cada um adota. O Axis2 adota como padrão seu próprio mecanismo de Data binding – o ADB –, enquanto o CXF implementa o padrão JAXB, que é o mesmo utilizado pela especificação JAX-WS. Por utilizar um padrão mais empregado, nesse aspecto, o CXF novamente se mostra superior ao Axis2;

7. **Criação de projetos através de *Archetypes* (Maven)**: O Axis2 não possui um *Archetype* nos repositórios oficiais do Maven. Durante os estudos foi encontrado um repositório não-oficial, porém, a utilização destes pode ser um

problema, pois depende de quem desenvolve o *Archetype* atualizar para as versões mais atuais do *framework* utilizado. Por outro lado, o CXF possui *Archetypes* nos repositórios oficiais do Maven e, portanto, o CXF se mostra superior ao Axis2;

8. **Ferramentas de apoio**: O Axis2 conta com uma série de ferramentas de apoio, principalmente para IDEs e geração de código, porém, o CXF apresenta um número maior de ferramentas e, reconhecidamente úteis para integração de sistemas;

9. **Formas de *deploy* de *Web services***: O *deploy* de todos os *Web services* desenvolvidos com Axis2 devem ser realizados em um aplicativo específico do Axis2. O *deploy* desse aplicativo, por sua vez, pode ser realizado em qualquer *container web* ou servidor de aplicação que suporte a extensão `.war`. Isso torna o *deploy* de *Web services* bem flexível, porém, as opções para criação de *Web services* não são tão flexíveis. A opção mais viável é a criação de um *Web service* com a extensão `.aar` e para isso é necessário utilizar um *plugin* para o Maven. Por outro lado, o *deploy* de um *Web service* construído com CXF também pode ser feito em vários tipos de *containers web* ou servidores de aplicação, com a vantagem de ser construído seguindo a especificação JAX-WS e não ser necessário um aplicativo extra, ou seja, o *deploy* pode ser feito diretamente. Nesse aspecto, mais uma vez, o CXF se mostra superior ao Axis2;

10. **Formas de desenvolvimento de consumidores**: Durante o desenvolvimento do consumidor com Axis2, utilizando AXIOM, notou-se que é uma tarefa complexa, principalmente pela necessidade de preencher um objeto `OMElement`. Já o CXF implementa consumidores de *Web services* baseados na especificação JAX-WS e possui opções mais produtivas. Nesse aspecto o CXF se mostrou também superior ao Axis2;

11. **Formas de depuração de erros**: Durante os estudos do Axis2 a única forma encontrada para realizar depuração de erros foi através de testes de unidade. Por outro lado, com o CXF foi possível configurar um *container web* diretamente na IDE e realizar o *deploy* do *Web service* e realizar a depuração de erros em um ambiente real. Logo, o CXF mostrou-se mais uma vez superior ao Axis2;

12. **Complexidade:** Levando em consideração a experiência do autor desta monografia com desenvolvimento de sistemas Java, pode-se afirmar que desenvolver *Web services* e consumidores com Axis2 é uma tarefa mais complexa do que com o CXF devido principalmente a: (i) Axis2 requer três arquivos de configuração para efetuar o *deploy* de um *Web service*, enquanto que o CXF requer apenas um arquivo de configuração e as demais informações são obtidas através de anotações Java; (ii) não seguir especificações consolidadas para processamento de documentos XML e *Data binding*, como adotado pelo CXF; e (iii) não possuir uma forma produtiva para depuração.

Em conclusão, o uso mais amplo do Axis2 se deve ao fato de ser um *framework* legado e, portanto, já ser adotado nas fábricas de software há mais tempo. No entanto, a avaliação comparativa dos *frameworks* permite concluir que o CXF tem diversas vantagens sobre o Axis2 – tais como implementação de um diversidade maior de especificações WS-*, seguir especificações já consolidadas, possuir suporte do Maven para criação de *Web services* através de *Archetypes*, possuir uma grande quantidade de ferramentas de apoio e também pelo fato do processo de desenvolvimento e implantação de *Web services* ser relativamente fácil – e, portanto, deveria ser o padrão atualmente adotado por fábricas de software.

REFERÊNCIAS

APACHE AXIS2. Disponível em: <http://axis.apache.org/axis2/java/core/>. Acesso em: 11 maio 2013.

APACHE CXF. Disponível em: <http://cxf.apache.org>. Acesso em: 27 abr. 2013.

API. Disponível em: <http://www.webopedia.com/TERM/A/API.html>. Acesso em: 16 mar. 2013.

CORBA FAQ. **OMG**. Disponível em: <http://www.omg.org/gettingstarted/ corbafaq.htm>. Acesso em: 02 jun. 2013.

BALANI, Naveen; HATHI, Rajeev. **Apache CXF Web Service Development.** Birmingham: Packt Publishing, 2009.

DEITEL, Harvey M. **Xml**: como programar. Tradução de Luiz Augusto Salgado e Edson Furmankiewicz. Porto Alegre: Bookman, 2003.

DEVMEDIA. **Padrão de Injeção de Dependência**. Disponível em: <http://www. devmedia.com.br/padrao-de-injecao-de-dependencia/18506>. Acesso em: 04 maio 2013.

DIKMANS, Lonneke; LUTTIKHUIZEN, Ronald van. **SOA Made Simple**. Birmingham: Packt Publishing, 2012.

ENDPOINT REFERENCE. **W3C**. Disponível em: <http://www.w3.org/Submission/ws-addressing/#_Toc77464317>. Acesso em: 17 abr. 2013.

ERL, Thomas. **SOA**: princípios de design de serviços. Tradução de Carlos Schafranski e Edson Furmankiewicz. São Paulo: Pearson, 2009.

FAYAD, M. E.; SCHMIDT, D. C. Object-oriented Application frameworks. **Communications of the ACM**. New York, p. 32-38, out. 1997.

FIELDING, T. Roy. **Architectural Styles and the Design of Network-based Software Architectures**. Disponível em: <http://www.ics.uci.edu/~fielding/pubs/ dissertation/top.htm>. Acesso em: 15 mar. 2013.

FOWLER, Martin. Disponível em: <http://www.martinfowler.com>. Acesso em: 30 maio 2013.

HAASE, Kim. **Java Message Service API Tutorial**.
Disponível em: <http://docs.oracle.com/javaee/1.3/jms/tutorial/jms_tutorial-1_3_1.pdf>. Acesso em: 27 maio 2013.

HOHPE Gregor; WOOLF Bobby. **Enterprise Integration Patterns:** Designing, Building, and Deploying Messaging Solutions, [S.I], Addison-Wesley Professional, 2003.

HTML INTRODUCTION. **W3SCHOOLS**. Disponível em: <http://w3schools.com/html/ html_intro.asp>. Acesso em: 01 abr. 2013.

JAVA BEANS. **ORACLE**. Disponível em: <http://docs.oracle.com/javase/tutorial/ javabeans/>. Acesso em: 23 abr. 2013.

JAVA BUSINESS INTEGRATION. **SearchSOA**. Disponível em: <http://searchsoa.techtarget.com/definition/Java-Business-Integration>. Acesso em: 30 abr. 2013.

JAVAWORLD. **XFire**: The easy and simple way to develop Web services Disponível em: <http://www.javaworld.com/javaworld/jw-05-2006/jw-0501-xfire.html>. Acesso em: 19 abr. 2013.

JAYASINGHE, Deepal; AZEEZ, Afkham. **Apache Axis2 Web Services**. Birmingham: 2. ed. Packt Publishing, 2011.

JCP. Disponível em: <http://www.jcp.org>. Acesso em: 17 abr. 2013.

KALIN, Martin. **Java Web Services**: Up and Running. O'Reilly Media, 2009.

KOCH, Christopher. **ABC da SOA**. Disponível em: <http://cio.uol.com.br/tecnologia /2006/07/17/idgnoticia.2006-07-17.3732358054/>. Acesso em: 14 mar. 2013.

MARSHALL, Dave. Remote Procedure Calls. Disponível em: <http://www.cs.cf.ac. uk/Dave/C/node33.html>. Acesso em: 13 mar. 2013.

MAVEN. Disponível em: <http://maven.apache.org>. Acesso em: 05 maio 2013.

MEPS. W3C. Disponível em: <http://www.w3.org/2002/ws/cg/2/07/meps.html>. Acesso em: 28 maio 2013.

OMG. Disponível em: <http://www.omg.org/gettingstarted/gettingstartedindex.htm>. Acesso em: 02 jun. 2013.

OPENSOURCE. Disponível em: <http://opensource.org/osd>. Acesso em: 23 abr. 2013.

PROCESSING INSTRUCTIONS. **W3C**. Disponível em: <http://www.w3.org/TR/REC-xml/#sec-pi>. Acesso em: 17 abr. 2013.

RICHARDSON, Michael. **Concept**: Enterprise Application Integration. Disponível em: <http://www.michael-richardson.com/rup_classic/modernize.legacy_evol/guidances/ concepts/enterprise_application_integration_3CEC2399.html>. Acesso em: 11 mar. 2013.

SERVICE-COMPONENT ARCHITECTURE. **SearchSOA**. Disponível em: <http://searchsoa.techtarget.com/definition/service-component-architecture>. Acesso em: 30 abr. 2013.

SOMMERVILLE, Ian. **Engenharia de Software**. Tradução de Maurício de Andrade. 6. ed. São Paulo: Pearson, 2003.

TANENBAUM, Andrew S.; WETHERALL, David. **Redes de Computadores**. Tradução de Daniel Vieira. 5. ed. São Paulo: Pearson, 2011.

TI EXPERT. **Annotations**. Disponível em: <http://www.tiexpert.net/programacao/java/annotations.php>. Acesso em: 23 abr. 2013.

W3C. Disponível em: <http://www.w3.org>. Acesso em: 17 abr. 2013.

W3SCHOOLS. Disponível em: <http://www.w3schools.com>. Acesso em: 01 abr. 2013.

APÊNDICE I – Implementação do sistema motivador

Este apêndice apresenta o código fonte de duas classes relevantes na implementação do sistema motivador: (i) `Partida`, que é a classe responsável por armazenar as informações relevantes de uma partida;

```
package br.com.ismael;

import java.util.GregorianCalendar;

/**
 * Classe que representa uma partida de futebol.
 * @author Ismael
 */
public class Partida {

  private String time1;
  private String time2;
  private String resultado;
  private String cidade;
  private String motivo;
  private String informacoesAdicionais;
  private GregorianCalendar data;

  public Partida(String time1, String time2, String resultado,
      GregorianCalendar data, String cidade, String motivo,
      String informacoesAdicionais) {
    this.time1 = time1;
    this.time2 = time2;
    this.data = data;
    this.resultado = resultado;
    this.cidade = cidade;
    this.motivo = motivo;
    this.informacoesAdicionais = informacoesAdicionais;
  }

  public Partida() {
  }

  /*
   * Métodos acessores para todos os atributos privados da classe.
   */
  @Override
  public String toString() {
    return this.getTime1() + " x " + this.getTime2() + ": " +
      this.getResultado() + " - " + this.getCidade() + " - "
        + this.getMotivo() + " - " + this.getInformacoesAdicionais();
  }

  public GregorianCalendar getData() {
    return data;
  }

  public void setData(GregorianCalendar data) {
    this.data = data;
  }
}
```

e (ii) `EfetuaConsultaPartida`, que é a classe que de fato efetua a consulta na base de dados. Por motivos de simplicidade, o código de consulta a um SGBD não foi incluído e, portanto, um objeto `Partida` com valores pré-fixados é sempre retornado.

```
1.  package br.com.ismael;
2.
3.  import java.util.GregorianCalendar;
4.
5.  /**
6.   * Classe que consulta o resultado de um jogo.
7.   * @author Ismael
8.   */
9.  public class EfetuaConsultaPartida {
10.
11.   /**
12.    * Através do nome dos times envolvidos e da cidade onde foi
13.    * realizado o confronto (opcional), retorna os dados
14.    * relevantes da partida, como resultado, motivo e informações
15.    * adicionais.
16.    *
17.    * @param nomeTime1
18.    * @param nomeTime2
19.    * @param data
20.    * @param cidade
21.    * @return Objeto {@link Partida} contendo as principais
22.    *  informações da partida requisitada.
23.    */
24.   public Partida consultaPartida(String time1, String time2,
25.       GregorianCalendar data, String cidade) {
26.     return new Partida(time1, time2, "2x1", data, cidade, "Amistoso",
27.         "Foi um jogo muito truncado, com muitas faltas e expulsões");
28.   }
29.
30. }
```

APÊNDICE II – Implementação Axis2

Este apêndice apresenta a classe `ClienteAxis2`, que representa a implementação de um consumidor de serviços com Axis2.

```
1.  package br.com.ismael;
2.  
3.  import java.awt.BorderLayout;
4.  import java.awt.GridLayout;
5.  import java.awt.event.ActionEvent;
6.  import java.awt.event.ActionListener;
7.  import java.util.Iterator;
8.  import javax.swing.JButton;
9.  import javax.swing.JFrame;
10. import javax.swing.JLabel;
11. import javax.swing.JOptionPane;
12. import javax.swing.JPanel;
13. import javax.swing.JTextField;
14. import javax.xml.datatype.DatatypeFactory;
15. import javax.xml.datatype.XMLGregorianCalendar;
16. import org.apache.axiom.om.OMAbstractFactory;
17. import org.apache.axiom.om.OMElement;
18. import org.apache.axiom.om.OMFactory;
19. import org.apache.axiom.om.OMNamespace;
20. import org.apache.axis2.Constants;
21. import org.apache.axis2.addressing.EndpointReference;
22. import org.apache.axis2.client.Options;
23. import org.apache.axis2.client.ServiceClient;
24. 
25. /**
26.  * Classe que representa um consumidor de web services feito com
27.  * Axis2.
28.  * @author Ismael
29.  */
30. public class ClienteAxis2 {
31. 
32.   private JFrame frame;
33.   private JPanel panel;
34.   private JPanel panelTimes;
35.   private JPanel panelResultado;
36.   private JLabel labelTime1;
37.   private JLabel labelTime2;
38.   private JLabel labelData;
39.   private JTextField fieldTime1;
40.   private JTextField fieldTime2;
41.   private JTextField fieldData;
42.   private JLabel labelCidade;
43.   private JTextField fieldCidade;
44.   private JButton consultar;
45.   private JButton cancelar;
46. 
47.   public ClienteAxis2() {
48.     this.frame = new JFrame("Consulta de partidas de futebol");
49.     this.panel = new JPanel();
50.     this.panelTimes = new JPanel();
51.     this.panelResultado = new JPanel();
52.     this.labelTime1 = new JLabel("Informe o time 1: ");
53.     this.fieldTime1 = new JTextField(30);
```

```
54.     this.labelTime2 = new JLabel("Informe o time 2: ");
55.     this.fieldTime2 = new JTextField(30);
56.     this.labelData = new JLabel("Informe a data da Partida: ");
57.     this.fieldData = new JTextField(10);
58.     this.labelCidade = new JLabel("Cidade (opcional): ");
59.     this.fieldCidade = new JTextField(30);
60.     this.consultar = new JButton("Consultar Partida");
61.     this.cancelar = new JButton("Cancelar");
62.   }
63.
64.   public void exibeInterfaceGrafica() {
65.     panel.setLayout(new BorderLayout());
66.     panelTimes.setLayout(new GridLayout(5, 2));
67.     panelTimes.add(labelTime1);
68.     panelTimes.add(fieldTime1);
69.     panelTimes.add(labelTime2);
70.     panelTimes.add(fieldTime2);
71.     panelTimes.add(labelData);
72.     panelTimes.add(fieldData);
73.     panelTimes.add(labelCidade);
74.     panelTimes.add(fieldCidade);
75.     consultar.addActionListener(new ActionListener() {
76.
77.       public void actionPerformed(ActionEvent e) {
78.         try {
79.           EndpointReference endPoint = new EndpointReference(
80.             "http://localhost:8080/axis2/services/ConsultaPartida");
81.           ClienteAxis2 clienteAxis2 = new ClienteAxis2();
82.           String[] dataSeparada = fieldData.getText().split("/");
83.           XMLGregorianCalendar dataGregorian = DatatypeFactory.
84.             newInstance().newXMLGregorianCalendar();
85.           dataGregorian.setDay(new Integer(dataSeparada[0]));
86.           dataGregorian.setMonth(new Integer(dataSeparada[1]));
87.           dataGregorian.setYear(new Integer(dataSeparada[2]));
88.           OMElement informacoesPartida = clienteAxis2.getPartida(
89.             "Atético-MG", "Cruzeiro", dataGregorian, null);
90.           Options options = new Options();
91.           options.setTo(endPoint);
92.           options.setTransportInProtocol(Constants.TRANSPORT_HTTP);
93.
94.           ServiceClient cliente = new ServiceClient();
95.           cliente.setOptions(options);
96.           OMElement resposta =
97.             cliente.sendReceive(informacoesPartida);
98.           Iterator<?> i = resposta.getChildElements();
99.           while (i.hasNext()) {
100.            Object object = i.next();
101.            StringBuffer buffer = new StringBuffer();
102.            buffer.append("Time 1: " + ((Partida)object).getTime1());
103.            buffer.append("\n");
104.            buffer.append("Time 2: " + ((Partida)object).getTime2());
105.            buffer.append("\n");
106.            buffer.append("Data: " + ((Partida) object).getData());
107.            buffer.append("\n");
108.            buffer.append("Resultado: " + ((Partida)object)
109.                  .getResultado());
110.            buffer.append("\n");
111.            buffer.append("Cidade: ");
112.            if (((Partida) object).getCidade() != null) {
113.              buffer.append(((Partida) object).getCidade());
114.            }
```

```
115.            buffer.append("\n");
116.            buffer.append("Motivo: " + ((Partida)object)
117.                  .getMotivo());
118.            buffer.append("\n");
119.            buffer.append("Informações Adicionais: " + ((Partida)
120.                  object).getInformacoesAdicionais());
121.            JOptionPane.showMessageDialog(null, buffer.toString(),
122.                  "RESULTADO DA CONSULTA",
123.                  JOptionPane.INFORMATION_MESSAGE);
124.          }
125.        } catch (Exception e2) {
126.          e2.printStackTrace();
127.        }
128.      }
129.    });
130.    panelTimes.add(consultar);
131.    cancelar.addActionListener(new ActionListener() {
132.
133.      public void actionPerformed(ActionEvent e) {
134.        System.exit(0);
135.
136.      }
137.    });
138.    panelTimes.add(cancelar);
139.    panel.add(panelTimes, BorderLayout.NORTH);
140.    panel.add(panelResultado, BorderLayout.SOUTH);
141.    frame.add(panel);
142.    frame.pack();
143.    frame.setLocationRelativeTo(null);
144.    frame.setVisible(true);
145.    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
146.  }
147.
148.  public OMElement getPartida(String nomeTime1, String nomeTime2,
149.      XMLGregorianCalendar data, String cidade) {
150.    OMFactory fac = OMAbstractFactory.getOMFactory();
151.    OMNamespace omNs = fac.createOMNamespace("http://ismael.com.br",
152.      "xs");
153.
154.    OMElement operacao = fac.createOMElement("consultaPartida",
155.      omNs);
156.    OMElement time1 = fac.createOMElement("nomeTime1", omNs);
157.    time1.addChild(fac.createOMText(time1, nomeTime1));
158.    operacao.addChild(time1);
159.
160.    OMElement time2 = fac.createOMElement("nomeTime2", omNs);
161.    time2.addChild(fac.createOMText(time2, nomeTime2));
162.    operacao.addChild(time2);
163.    OMElement dataPartida = fac.createOMElement("data", omNs);
164.    dataPartida.addChild(fac.createOMText(dataPartida,
165.      data.toXMLFormat()));
166.    operacao.addChild(dataPartida);
167.    OMElement cidadePartida = fac.createOMElement("cidade", omNs);
168.    cidadePartida.addChild(fac.createOMText(cidadePartida, cidade));
169.    operacao.addChild(cidadePartida);
170.
171.    return operacao;
172.  }
173.
174.  public static void main(String[] args) {
175.
```

```
176.      ClienteAxis2 cliente = new ClienteAxis2();
177.      cliente.exibeInterfaceGrafica();
178.
179.    }
180.
181.    /*
182.     * Métodos acessores para todos os atributos privados da classe.
183.     */
184.
185.  }
```

APÊNDICE III – Implementação CXF

Este apêndice apresenta o código fonte de duas classes e um documento XML que representa um arquivo de configuração: (i) `ConsultaPartidaImpl`, representa a implementação de um *Web service* com CXF;

```
1.  package br.com.ismael;
2.
3.  import java.util.GregorianCalendar;
4.  import javax.jws.WebService;
5.  import br.com.ismael.Partida;
6.
7.  @WebService(endpointInterface = "br.com.ismael.ConsultaPartida")
8.  public class ConsultaPartidaImpl implements ConsultaPartida {
9.
10.   @Override
11.   public Partida consultaPartida(String nomeTime1, String nomeTime2,
12.       GregorianCalendar data, String cidade) {
13.     return new EfetuaConsultaPartida().consultaPartida(nomeTime1,
14.       nomeTime2, data, cidade);
15.   }
16.
17. }
```

(ii) `ClienteCXF`, que representa um consumidor de serviços com CXF;

```
1.  package br.com.ismael;
2.
3.  import java.awt.BorderLayout;
4.  import java.awt.GridLayout;
5.  import java.awt.event.ActionEvent;
6.  import java.awt.event.ActionListener;
7.  import javax.swing.JButton;
8.  import javax.swing.JFrame;
9.  import javax.swing.JLabel;
10. import javax.swing.JOptionPane;
11. import javax.swing.JPanel;
12. import javax.swing.JTextField;
13. import javax.xml.datatype.DatatypeFactory;
14. import javax.xml.datatype.XMLGregorianCalendar;
15. import org.apache.cxf.endpoint.Client;
16. import org.apache.cxf.endpoint.dynamic.DynamicClientFactory;
17.
18. /**
19.  * Classe que invoca um método de um web service de forma dinâmica
20.  * através de um WSDL previamente publicado.
21.  * @author Ismael
22.  */
23. public class ClienteCXF {
24.
25.   private JFrame frame;
26.   private JPanel panel;
27.   private JPanel panelTimes;
28.   private JPanel panelResultado;
```

```
29.    private JLabel labelTime1;
30.    private JLabel labelTime2;
31.    private JLabel labelData;
32.    private JTextField fieldTime1;
33.    private JTextField fieldTime2;
34.    private JTextField fieldData;
35.    private JLabel labelCidade;
36.    private JTextField fieldCidade;
37.    private JButton consultar;
38.    private JButton cancelar;
39.
40.    public ClienteCXF() {
41.      this.frame = new JFrame("Consulta de partidas de futebol");
42.      this.panel = new JPanel();
43.      this.panelTimes = new JPanel();
44.      this.panelResultado = new JPanel();
45.      this.labelTime1 = new JLabel("Informe o time 1: ");
46.      this.fieldTime1 = new JTextField(30);
47.      this.labelTime2 = new JLabel("Informe o time 2: ");
48.      this.fieldTime2 = new JTextField(30);
49.      this.labelData = new JLabel("Informe a data da Partida: ");
50.      this.fieldData = new JTextField(10);
51.      this.labelCidade = new JLabel("Cidade (opcional): ");
52.      this.fieldCidade = new JTextField(30);
53.      this.consultar = new JButton("Consultar Partida");
54.      this.cancelar = new JButton("Cancelar");
55.    }
56.
57.    public void exibeInterfaceGrafica() {
58.      panel.setLayout(new BorderLayout());
59.      panelTimes.setLayout(new GridLayout(5, 2));
60.      panelTimes.add(labelTime1);
61.      panelTimes.add(fieldTime1);
62.      panelTimes.add(labelTime2);
63.      panelTimes.add(fieldTime2);
64.      panelTimes.add(labelData);
65.      panelTimes.add(fieldData);
66.      panelTimes.add(labelCidade);
67.      panelTimes.add(fieldCidade);
68.      consultar.addActionListener(new ActionListener() {
69.
70.        @Override
71.        public void actionPerformed(ActionEvent e) {
72.          DynamicClientFactory factory = DynamicClientFactory.
73.              newInstance();
74.          // Cria o cliente através do endereço do WSDL
75.          Client client = factory.createClient(
76.           "http://localhost:8080/webservicecxf/ConsultaPartida?wsdl");
77.          try {
78.            /*
79.             * Invoca o serviço através do nome da operação e
80.             * informa os parâmetros necessários para a execução do
81.             * mesmo
82.             */
83.            String[] dataSeparada = fieldData.getText().split("/");
84.            XMLGregorianCalendar dataGregorian = DatatypeFactory.
85.              newInstance().newXMLGregorianCalendar();
86.            dataGregorian.setDay(new Integer(dataSeparada[0]));
87.            dataGregorian.setMonth(new Integer(dataSeparada[1]));
88.            dataGregorian.setYear(new Integer(dataSeparada[2]));
89.            Object[] obj = client.invoke("consultaPartida",
```

```
90.              fieldTime1.getText(), fieldTime2.getText(),
91.              dataGregorian, fieldCidade.getText());
92.          for (Object object : obj) {
93.            StringBuffer buffer = new StringBuffer();
94.            buffer.append("Time 1: " + ((Partida)
95.                  object).getTime1());
96.            buffer.append("\n");
97.            buffer.append("Time 2: " + ((Partida)
98.                  object).getTime2());
99.            buffer.append("\n");
100.           buffer.append("Data: " + ((Partida) object).getData());
101.           buffer.append("\n");
102.           buffer.append("Resultado: " + ((Partida)
103.                 object).getResultado());
104.           buffer.append("\n");
105.           buffer.append("Cidade: ");
106.           if (((Partida) object).getCidade() != null) {
107.             buffer.append(((Partida) object).getCidade());
108.           }
109.           buffer.append("\n");
110.           buffer.append("Motivo: " + ((Partida)
111.                 object).getMotivo());
112.           buffer.append("\n");
113.           buffer.append("Informações Adicionais: " + ((Partida)
114.                 object).getInformacoesAdicionais());
115.           JOptionPane.showMessageDialog(null, buffer.toString(),
116.                 "RESULTADO DA CONSULTA",
117.               JOptionPane.INFORMATION_MESSAGE);
118.         }
119.       } catch (Exception e1) {
120.         e1.printStackTrace();
121.       }
122.     }
123.   });
124.   panelTimes.add(consultar);
125.   cancelar.addActionListener(new ActionListener() {
126.
127.     @Override
128.     public void actionPerformed(ActionEvent e) {
129.       System.exit(0);
130.
131.     }
132.   });
133.   panelTimes.add(cancelar);
134.   panel.add(panelTimes, BorderLayout.NORTH);
135.   panel.add(panelResultado, BorderLayout.SOUTH);
136.   frame.add(panel);
137.   frame.pack();
138.   frame.setLocationRelativeTo(null);
139.   frame.setVisible(true);
140.   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
141. }
142.
143. public static void main(String[] args) {
144.
145.   ClienteCXF cliente = new ClienteCXF();
146.   cliente.exibeInterfaceGrafica();
147.
148. }
149.
150. /*
```

```
151.     * Métodos acessores para todos os atributos privados da classe.
152.     */
153.
154. }
```

e (iii) `beans.xml`, que representa o arquivo de configuração baseado no Spring *framework*, necessário para publicação de um *Web service* com CXF.

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <beans xmlns="http://www.springframework.org/schema/beans"
3.  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4.    xmlns:jaxws="http://cxf.apache.org/jaxws"
5.    xsi:schemaLocation="
6.        http://www.springframework.org/schema/beans
7.        http://www.springframework.org/schema/beans/spring-beans.xsd
8.        http://cxf.apache.org/jaxws
9.        http://cxf.apache.org/schemas/jaxws.xsd">
10.
11.   <import resource="classpath:META-INF/cxf/cxf.xml" />
12.
13.   <jaxws:endpoint id="consultaJogo" implementor=
14.     "br.com.ismael.ConsultaPartidaImpl" address="/ConsultaPartida" />
15.
16. </beans>
```