NTNU
Innovation and Creativity

Input Validation Framework for Web Services

Henning Jensen

Master of Science in Computer Science
Submission date: May 2007
Supervisor: Torbjørn Skramstad, IDI
Co-supervisor: Carl Christensen, BEKK Consulting AS

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The use of Web Services has grown rapidly the last years. Integrating existing systems and building many small systems that collaborate through Web Services to form a larger system, is becoming more and more important in the software industry. Unfortunately the focus of the earliest Web Service standards was on functionality and not security. This has, however, improved lately with the release of several security standards to support e.g. confidentiality and integrity.

The master thesis «Adding Security to Web Services» by Brekken and Åsprang, proposed different approaches for creating secure Web Services. Input Validation of SOAP envelopes was realized with XML Schemas. This project will build upon the experiences and results found in that thesis to develop a framework for input validation in Web Services.

The goals of this project are:
* Define and develop a framework for using, developing and maintaining Web Service input validation policies in a structured, distributable and reusable manner.
* Demonstrate how to utilize and integrate the framework with existing SOAP frameworks.

Assignment given: 19. January 2007
Supervisor: Torbjørn Skramstad, IDI

Abstract

Security is an important aspect for all kinds of software development, but it is especially important for web applications since they usually are exposed to the Internet. Web Services offer application to application connectivity using the SOAP protocol. Web Services are quite often built as an extension to already existing applications to provide business to business communication. Since it is often necessary to expose critical business functions through the Web Services, e.g., ordering an item or sending an invoice, security in Web Services are vital for a company's daily operations.

In this project we have created an input validation framework for Web Services, to aid developers in creating more secure Web Services in an easier and more reusable manner. We have focused on creating a lightweight policy configuration based on XML, and a set of highly configurable and extendable validators. The framework is implemented in Java and is not dependent on a specific SOAP framework. To keep the framework general and compatible with multiple SOAP frameworks, we have developed a set of interceptors to support the two most common open source SOAP frameworks, Codehaus XFire and Apache Axis2.

This report first presents theory and rationale behind the need for a new way of performing input validation. Further the implementation of the framework is documented together with an example application, which demonstrates an example use of the framework.

Preface

This master's thesis documents the work done by Henning Jensen from January to June 2007. It is the final work on a Master of Technology degree from the Department of Computer Science and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). The assignment was initiated by Bekk Consulting AS (BEKK) as a part of their involvement in the open source community.

I would like to thank Professor Torbjørn Skramstad at IDI/NTNU and Carl C. Christensen at Bekk Consulting AS for all their guidance and valuable feedback during the project. I really appreciate their time and efforts.

June 1, 2007

Henning Jensen

Contents

List of Figures

List of Tables

List of Listings

List of abbreviations

API Application Programming Interface

AXIOM AXis Object Model

DTD Document Type Definition

REST REpresentational State Transfer

SOAP Simple Object Access Protocol

StAX Streaming API for XML

TDD Test Driven Development

UDDI Universal Description Discovery & Integration

URL Uniform Resource Locator

XML Extensible Markup Language

W3C World Wide Web Consortium

WSDL Web Services Description Language

WSPV Web Service Payload Validator

XP eXtreme Programming

Chapter 1

Introduction

This chapter presents the motivational background and the problem definition, including the goals, for the project. We then introduce the project context and the report outline.

Motivation

Input validation is one of the most important aspects in web application security. Being able to trust your users' input is important in order to allow them accessing critical business systems.

Web applications are a bit different than regular desktop software. They are harder to keep secure since they are accessible to anyone with a connection to the Internet. A web application encourages user mobility, but at the same time it is exposed and accessible to anyone with Internet access.

Symantec's Internet Security Threat Report [1] is published every six months. In volume XI, covering the period between July 1 and December 31 2006, 66 percent of the reported vulnerabilities affected web applications. The second most common cause of data breaches was insecure policies, which made up 28 percent of all incidents. These numbers emphasize the importance of properly secured Web Services.

The growing number of web applications and the need for interoperability laid the foundation for Web Services (SOAP) [2]. Web Services allow for systems to interact with XML over HTTP. Applications can exchange data between themselves independently of platform and programming language. Since data is going in and out of the system boundaries, input validation is needed to keep the system secure. Input validation in Web Services has, however, not been very easy to comprehend. The developers have usually been left alone with the security issues without much help from the SOAP frameworks. A framework that aids the developer performing input validation in a structured and reusable manner, could contribute to making Web Services more secure.

Problem Definition

The master thesis *Adding Security to Web Services* [3] proposed different approaches for creating secure Web Services. Input Validation of SOAP envelopes was realized with XML Schemas. This project will build upon the experiences and results found in that thesis to develop a framework for input validation in Web Services.

The goals of this project are:

- Define and develop a framework for using, developing and maintaining Web Service input validation policies in a structured, distributable and reusable manner.
- Demonstrate how to utilize and integrate the framework with existing SOAP frameworks.

Project Context

This project is a master's thesis carried out at the Software Engineering Group which is a part of the Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU) in Trondheim. The project is also carried out in cooperation with Bekk Consulting AS (BEKK), which is a leading Norwegian business consulting and technology service company.

Report Outline

The report has been split into four major chapters. Each chapter is briefly introduced below.

Introduction: Gives a brief overview of the work, the problem definition, its motivational background, the project context and this report outline.

Theoretical Background: Contains the necessary literature study performed in the earlier stages of the project. This forms the basis for the requirements and choices made later in the project. This chapter covers four major topics: XML, Web Services, SOAP frameworks and security in Web Services.

Contribution: This chapter contains the contribution work for this thesis. We will present the issues we have found with today's most common method on performing input validation, before we move on with the requirements and architecture of our input validation framework. Then we will cover the details of the implementation. The last section contains an example application, which is used to demonstrate the usage of the framework.

Evaluation and Conlusion: Contains a discussion and evaluation of the project together with the conclusion. At the end there is a section about further work and suggestions for the framework.

Chapter 2

Theoretical Background

This chapter presents background information around the topics of Web Services. A brief introduction to XML will be given, and a more thorough explanation of the concept of Web Services. We will also present an overview of the two most common frameworks used to create Web Services in Java. Security concerns in Web Services will be covered in the last section of the chapter.

2.1 XML

This section will give a brief introduction to XML. For a more thorough introduction we recommend [4, 5].

Extensible Markup Language (XML) [6] was introduced by the World Wide Web Consortium (W3C) in 1998. Two kinds of XML exists, document-centric and data-centric. Document-centric XML origins from SGML and has been used to represent semi structured documents such as technical manuals and product catalogs. The key factor in these documents is the semi structured mark-up text [4].

```
<h1>Product details for Samsung Syncmaster 225BW</h1>
<p>LCD-screen details:</p>

<list>
 <item>Size: 22’</item>
 <item>Latency: 5ms</item>
 <item>Contrast: 700:1</item>
</list>
```

Listing 2.1: Document-centric XML

Listing 2.1 shows an example of document-centric XML. Document-centric XML is very human-friendly since the document structure is similar to a top-down plain text document. The document begins with a header, then a paragraph and ends with a list of properties. The markup does not interfere with the structure of the document.

Data-centric XML is used to create highly structured information, i.e. textual representation of relational data from databases, financial transaction information or programming language data structures. Data-centric XML is often generated by machines and meant for machine consumption [4], even though it usually is quite readable for humans also. Listing 2.2 shows an example of data-centric XML.

```
<links>
 <category name="Newspapers">
  <link title="The New York Times">http://nytimes.com/</link>
  <link title="International Herald Tribune">http://www.iht.com/</link>
 </category>
 <category name="Personal">
  <link title="John’s Homepage">http://myspace.com/johndoe</link>
 </category>
</links>
```

Listing 2.2: Data-centric XML

The structure of an XML document is quite simple since it just contains one type of logical structures: Elements. An element can contain other elements or it can hold a concrete value. Elements can also have attributes assigned to them. Attributes are name-value pairs that belong to a certain element and is often used to provide uniqueness, type constraints or setting default values.

2.1.1 Namespaces

An important property of XML documents is their ability to form large documents composed by several small documents. This makes it possible to reuse XML, but at the same time this creates a new problem: *name collisions*. If two elements with the same name from two different XML documents are joined together to form a single document, you will not have any way of uniquely identifying the two elements.

This is why the *Namespace in XML* standard [7] was created. This standard defines how to enable different namespaces for different parts of an XML document. The namespace helps creating a unique identifier for all elements, making it possible to correctly process a document without having to worry about name collisions. Listing 2.3 shows an example of the use of namespaces. The name *edi* refers to a unique namespace that makes the price element unique for the rest of the document.

```
<edi:price xmlns:edi="http://ecommerce.example.org/schema" units="Euro">
 32.18
</edi:price>
```

Listing 2.3: Namespaces in XML

Namespaces are widely used in Web Services and more complex XML structures. In small and simple XML documents you rarely have to worry about namespaces.

2.1.2 XML Schema

The predecessor of XML Schema is Document Type Definition (DTD)[6]. DTDs has only support for specifying the most simple part of an XML document, the structure. It lacks support for namespace integration, modular vocabulary design and flexible content models. To address these issues the W3C came up with the XML Schema standard [8]. XML Schema is a standard on how to validate the structure and content of an XML document. The schema contains information about which elements a document can contain and their constraints.

Listing 2.4 shows an example of an XML document. The corresponding XML schema is shown in Listing 2.5. Examples from [9].

```
<purchaseOrder orderDate="1999-10-20">
   <shipTo country="US">
      <name>Alice Smith</name>
      <street>123 Maple Street</street>
      <city>Mill Valley</city>
      <state>CA</state>
      <zip>90952</zip>
   </shipTo>
   <billTo country="US">
      <name>Robert Smith</name>
      <street>8 Oak Avenue</street>
      <city>Old Town</city>
      <state>PA</state>
```

```
      <zip>95819</zip>
   </billTo>
   <comment>Hurry, my lawn is going wild</comment>
   <items>
      <item partNum="872-AA">
         <productName>Lawnmower</productName>
         <quantity>1</quantity>
         <USPrice>148.95</USPrice>
         <comment>Confirm this is electric</comment>
      </item>
      <item partNum="926-AA">
         <productName>Baby Monitor</productName>
         <quantity>1</quantity>
         <USPrice>39.98</USPrice>
         <shipDate>1999-05-21</shipDate>
      </item>
   </items>
</purchaseOrder>
```

Listing 2.4: XML Schema - Document Example

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

  <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>

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

  <xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
      <xsd:element name="shipTo" type="USAddress"/>
      <xsd:element name="billTo" type="USAddress"/>
      <xsd:element ref="comment" minOccurs="0"/>
      <xsd:element name="items"  type="Items"/>
    </xsd:sequence>
    <xsd:attribute name="orderDate" type="xsd:date"/>
  </xsd:complexType>

  <xsd:complexType name="USAddress">
    <xsd:sequence>
      <xsd:element name="name"   type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city"   type="xsd:string"/>
      <xsd:element name="state"  type="xsd:string"/>
      <xsd:element name="zip"    type="xsd:decimal"/>
    </xsd:sequence>
    <xsd:attribute name="country" type="xsd:NMTOKEN"
                   fixed="US"/>
  </xsd:complexType>

  <xsd:complexType name="Items">
    <xsd:sequence>
      <xsd:element name="item" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="productName" type="xsd:string"/>
            <xsd:element name="quantity">
              <xsd:simpleType>
                <xsd:restriction base="xsd:positiveInteger">
                  <xsd:maxExclusive value="100"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
```

```
            <xsd:element name="USPrice"  type="xsd:decimal"/>
            <xsd:element ref="comment"   minOccurs="0"/>
            <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
          </xsd:sequence>
          <xsd:attribute name="partNum" type="SKU" use="required"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>

  <!-- Stock Keeping Unit, a code for identifying products -->
  <xsd:simpleType name="SKU">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{3}-[A-Z]{2}"/>
    </xsd:restriction>
  </xsd:simpleType>

</xsd:schema>
```

Listing 2.5: XML Schema - Schema Example

We can see that the schema language is strong. It can handle complex data structures, e.g. objects with other objects as properties, simple data types e.g. *xsd:string*, *xsd:date*, *xsd:positiveInteger*, and various restrictions e.g. patterns, minOccurs, nillable. All of these properties makes the XML Schema language powerful, even though it is quite verbose. If we were to write the same definition using DTD we would have a smaller but also less powerful validation schema. Listing 2.6 shows an example of the same schema written in DTD.

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT purchaseOrder (shipTo,billTo,comment,items)>
<!ATTLIST purchaseOrder
        orderDate CDATA #REQUIRED
>
<!ELEMENT shipTo (name,street,city,state,zip)>
<!ATTLIST shipTo
        country CDATA #REQUIRED
>
<!ELEMENT billTo (name,street,city,state,zip)>
<!ATTLIST billTo
        country CDATA #REQUIRED
>
<!ELEMENT name (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
<!ELEMENT comment (#PCDATA)>
<!ELEMENT items (item+)>
<!ELEMENT item (productName,quantity,USPrice,shipDate?,comment?)>
<!ATTLIST item
        partNum CDATA #REQUIRED
>
<!ELEMENT productName (#PCDATA)>
<!ELEMENT quantity (#PCDATA)>
<!ELEMENT USPrice (#PCDATA)>
<!ELEMENT shipDate (#PCDATA)>
<!ELEMENT comment (#PCDATA)>
```

Listing 2.6: XML Schema - Corresponding DTD

DTDs does not have data types, and this is probably one of the most common reasons why developers are moving from DTD to XML Schema.

[10] investigated the use of XML Schemas versus DTD. They found that 73% of their collected XML Schemas used the simple type restrictions. The rest of the functionality was practically unused. If this is due to the complexity or immaturity of the language is hard to figure out, but it indicates that XML Schemas might be a bit complex for the average developer.

In fact both [10, 11] concludes that most of the XML Schemas could have been written in DTD from the structure point of view. This is an important factor as we will introduce Web Services, where XML Schemas are the primary tool for performing input validation.

2.2 Web Services

A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web Service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. (World Wide Web Consortium) [12]

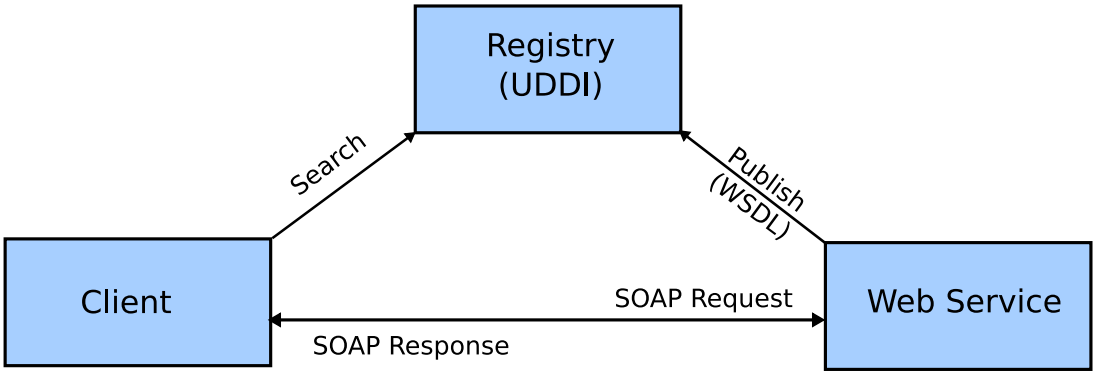

Figure 2.1: Web Services Overview

Web Services is based on normal client-server architecture. Figure 2.1 shows a conceptual overview of how clients interact with a Web Service. The client and server communicate by sending SOAP messages (XML) directly between themselves. Before this communication takes place the client needs to be aware of what services are offered, and how they can be utilized. This is done through a Web Services Description Language (WSDL) document. This document can be published directly to the client or looked up via a registry service called Universal Description Discovery & Integration (UDDI). In the next sections we will give an introduction to the SOAP protocol and WSDL-documents.

2.2.1 Simple Object Access Protocol (SOAP)

A Web Service is normally invoked by sending a request to a HTTP url (note that other transportation mechanisms than HTTP can be used, e.g. SMTP). SOAP is the message protocol used for communicating with Web Services. This protocol is based on XML, and a message is often referred to as an envelope. This envelope consists of a header and a body element. Figure 2.2 illustrates the structure of a SOAP envelope. The envelope body must contain an element with the name of the operation (method) to invoke. All arguments to the operation call must reside inside this element. This content of the envelope body is often referred to as the *payload* of the envelope, as it is the actual message from the client to the server.

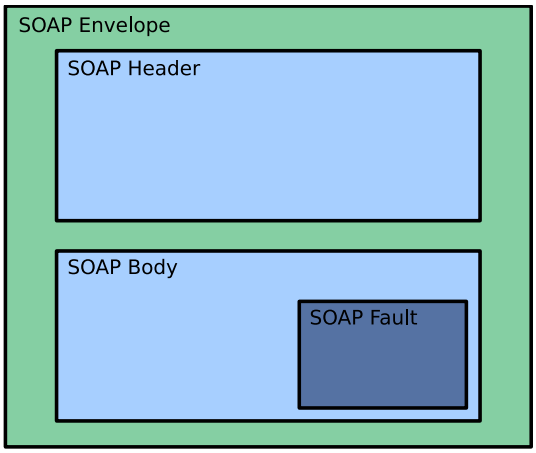

Figure 2.2: SOAP Envelope Structure

An example of a SOAP request and response envelope is shown in Listings 2.7 and 2.8 respectively. Note that namespaces (2.1.1) to the elements inside the envelope body have been removed to increase readability. In this exmple the client is sending a request to the *findBook* operation with an isbn number as a parameter. The server then generates a response containing a *Book* object.

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soap:Header/>
  <soap:Body>
    <findBook xmlns="http://wspv.fjas.no/BookService">
      <isbn>
        12-3456-789-0
      </isbn>
    </findBook>
  </soap:Body>
</soap:Envelope>
```

Listing 2.7: SOAP Request Envelope

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soap:Header/>
  <soap:Body>
    <findBookResponse xmlns="http://wspv.fjas.no/BookService">
      <book xmlns="http://wspv.fjas.no/BookService">
        <author>Dan Diephouse</author>
        <id>1</id>
        <isbn>12-3456-789-0</isbn>
        <price>49.99</price>
        <title>Using XFire</title>
      </book>
    </findBookResponse>
  </soap:Body>
</soap:Envelope>
```

Listing 2.8: SOAP Response Envelope

The SOAP protocol allows extensions through the use of the header element. The header can contain extra information that is processed by the receiver. Many Web Service extension standards have been released, the most prominent security standards are listed in Table 2.2.1.

Name	**Purpose**
WS-Addressing	Enables messaging systems to support message transmission through networks that include processing nodes such as endpoint managers, firewalls, and gateways in a transport-neutral manner. [13]
WS-Policy	Defines a base set of constructs that can be used and extended by other Web Services specifications to describe a broad range of service requirements and capabilities. [14]
WS-Security	Provides SOAP message integrity and confidentiality. Can be use to accommodate a wide variety of security models and encryption technologies. [15]
WS-SecurityPolicy	Policy assertions for use with WS-Policy which apply to WSS: SOAP Message Security, WS-Trust and WS-SecureConversation. [16]
WS-Reliability	Protocol for exchanging SOAP messages with guaranteed delivery, no duplicates, and guaranteed message ordering. [17]

Table 2.1: Web Service Extension Standards

Even though there are several published security extensions like WS-Security [15] and WS-SecurityPolicy [16], none of them can be said to directly handle issues with proper validation of input data. This is usually handled indirectly by the WSDL document, which we will cover in Section 2.2.2.

SOAP Faults are a special element in the SOAP protocol [2]. A SOAP Fault can carry error information and is returned to the client if a server side error occurs during

processing of a request. It can carry several special elements inside, like *code*, *reason*, *node* and *detail*, to give detailed information to the client. This is, however, something that should be used with precaution. Giving out too much error information can be a huge security risk [18]. The SOAP Fault should therefore be used with caution, and be as minimalistic and non-informative as possible when used in a production system.

2.2.2 Web Services Description Language (WSDL)

WSDL is used, as the abbreviation indicates, to describe a Web Service. In order for a client to be able to use a Web Service, it needs to know how to use it. Which operations are available and what kind of arguments each operation requires is vital information, and this is what the WSDL-document describes.

Web Services are described using six major elements (from the WSDL specification [19]):

- **types**, which provides data type definitions used to describe the messages exchanged.
- **message**, which represents an abstract definition of the data being transmitted. A message consists of logical parts, each of which is associated with a definition within the type system.
- **portType**, which is a set of abstract operations. Each operation refers to an input message and output message.
- **binding**, which specifies concrete protocol and data format specifications for the operations and messages defined by a particular portType.
- **port**, which specifies an address for a binding, thus defining a single communication endpoint.
- **service**, which is used to aggregate a set of related ports.

In order to explain WSDL further we will look closer at an example WSDL document that describes the findBook-operation invoked in the previous SOAP envelope example in Listing 2.7. Listing 2.9 shows the *types* definition needed for the *findBook* operation which the SOAP envelopes in Listing 2.7 and 2.8 interact with.

```
<wsdl:types>
   <xsd:schema targetNamespace="http://wspv.fjas.no/BookService" elementFormDefault=
       "qualified" attributeFormDefault="qualified">
     <!-- define book object -->
     <xsd:complexType name="Book">
       <xsd:sequence>
         <xsd:element name="author" type="xsd:string" minOccurs="0" nillable="true"
             />
         <xsd:element name="id" type="xsd:int" minOccurs="0" />
         <xsd:element name="isbn" type="xsd:string" minOccurs="0" nillable="true" />
         <xsd:element name="price" type="xsd:double" minOccurs="0" />
         <xsd:element name="title" type="xsd:string" minOccurs="0" nillable="true" /
             >
       </xsd:sequence>
```

```
    </xsd:complexType>
    <!-- define book exception -->
    <xsd:complexType name="BookExceptionDetail">
      <xsd:sequence>
        <xsd:element name="code" type="xsd:string" minOccurs="0" nillable="true" />
        <xsd:element name="detailMessage" type="xsd:string" minOccurs="0" nillable=
            "true" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:schema>
  <xsd:schema targetNamespace="http://wspv.fjas.no/BookService" elementFormDefault=
      "qualified" attributeFormDefault="qualified">
    <!-- define findBook method -->
    <xsd:element name="findBook">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="isbn" type="xsd:string" nillable="true" minOccurs="1"
              maxOccurs="1" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <!-- define the findBook response -->
    <xsd:element name="findBookResponse">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="book" type="ns1:Book" nillable="true" minOccurs="1"
              maxOccurs="1" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
  <xsd:schema targetNamespace="http://wspv.fjas.no/BookService" elementFormDefault=
      "qualified" attributeFormDefault="qualified">
    <xsd:element name="BookFault" type="ns1:BookExceptionDetail" />
  </xsd:schema>
</wsdl:types>
```

Listing 2.9: WSDL Document - Type definitions

This *types* definition is a good example of the verbosity of the WSDL-document. To properly define the *findBook* request with an isbn number and the response returning a book object with five properties, a total of 38 lines of xml is needed. The important thing to notice here is that the *types* definitions is built up by using XML Schema (see 2.1.2). This means that the datatypes to be sent in and out of a Web Service can be constrained according to the XML Schema standard [8]. Note that you can not trust the data unless you are 100% sure that the SOAP framework supports and enforces all specified constraints.

Moving on with the *message* definition in the WSDL document, shown in listing 2.10, we can see that it defines three types of messages: *findBookRequest*, *findBookResponse* and *BookFault*. Each message contain a *part* element which points back to the elements in the *types* definitions we saw in the Listing 2.9.

```
<wsdl:message name="findBookRequest">
  <wsdl:part element="tns:findBook" name="parameters" />
</wsdl:message>,
<wsdl:message name="findBookResponse">
  <wsdl:part element="tns:findBookResponse" name="parameters" />
</wsdl:message>
<wsdl:message name="BookFault">
  <wsdl:part element="ns2:BookFault" name="BookFault" />
</wsdl:message>
```

Listing 2.10: WSDL Document - Message definitions

A *port type* refer to a complete operation including input message, output message and a potential fault message. Listing 2.11 defines the *findBook* operation which consists of all of the three messages defined in the previous Listing, 2.10.

```
<wsdl:portType name="BookServicePortType">
  <wsdl:operation name="findBook">
    <wsdl:input message="tns:findBookRequest" name="findBookRequest" />
    <wsdl:output message="tns:findBookResponse" name="findBookResponse" />
    <wsdl:fault message="tns:BookFault" name="BookFault" />
  </wsdl:operation>
</wsdl:portType>
```

Listing 2.11: WSDL Document - Port Type definitions

The last element, the *binding* definition, refers to the protocol and data format of all of the messages. Messages can be encoded in various ways to support different communication methods. SOAP is not necessary only transported over HTTP, it can also be sent with SMTP and other protocols, but HTTP transport is the most common.

2.2.3 Universal Description Discovery & Integration (UDDI)

UDDI is used for automatic discovery and publishing of WSDL documents. The rationale behind the development of UDDI was to enable clients to automatically look up Web Services in a central registry, just like the yellow pages in the phone book. A client can browse through the registry and find a suitable service. In the registry lies technical information about the service and its interfaces (WSDL-document) for the client to use.

Since we are focusing on input validation and security, UDDI is out of scope for this project. We recommend [20] for more information about UDDI.

2.3 SOAP Frameworks

In Java we need an application server or servlet container when running a web application. To run Web Services as well, we need a SOAP framework. The SOAP framework is responsible for receiving and sending SOAP messages to the client.

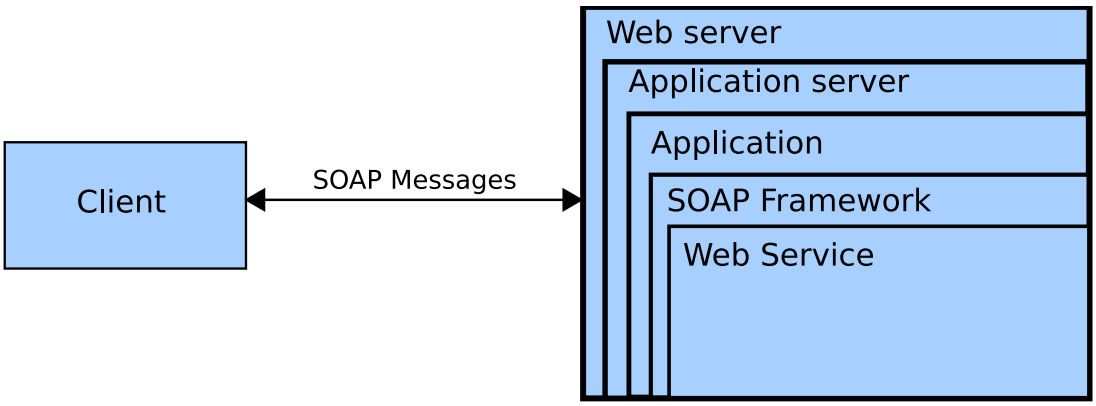

Figure 2.3: SOAP Frameworks Overview

Figure 2.3 shows where SOAP frameworks operate. The SOAP framework is integrated within the running application and intercepts all SOAP requests that are sent to the application. This is done by reserving an Uniform Resource Locator (URL) for SOAP requests only, e.g. */services/**. This URL serves as a basis endpoint for all Web Services. For the SOAP framework to identify which service the client can use, each service is assigned an unique identifier. E.g., if we have a *BookService* with a *findBook* operation that allows a client to search for books, the address for this service could be *http://www.example.com/services/BookService*

When the SOAP envelope is intercepted, the headers are inspected and various operations are performed before the correct internal service in the application receives the request. The SOAP envelope headers can contain authentication tokens, encryption information and other information that the SOAP framework needs to take care of before the envelope is forwarded. Most frameworks also support XML binding, which means it transforms the payload[1] of the SOAP envelopes into objects, that are suitable for interacting with underlying services within the application.

Two popular open source SOAP frameworks are *Apache Axis 2* and *XFire*. Their key features and architecture will be presented in the next sections.

[1]Payload is here referred to as the body content of the envelope (see 2.2.1)

2.3.1 Apache Axis2

Axis2 is an implementation of the W3C SOAP Recommendation [2]. The project is a follow-up on the original Apache SOAP project [21], which started up in june 2000. The Axis project has been one of the most successful open source SOAP frameworks with its large user community, and the wide range of applications using it.

Some of the key features in Axis2 are [22]:

- Streaming API for XML (StAX) - Fast API for XML Processing
- AXis Object Model (AXIOM) - Light-weight XML object model
- Hot Deployment - Runtime deployment of new services
- Component-oriented Deployment - Plugin processing handlers
- WSDL 1.1 and 2.0 support
- REpresentational State Transfer (REST) support [23]

Axis2s architecture is built with flexibility and modularity in mind. The core engine is minimalistic and extra features are supported by pluggable modules. Figure 2.4 shows the general component architecture of Axis2. The foundation of Axis2 is the StAX and AXIOM API. On top of this is the core with the pluggable modules and the deployment functionality. Axis2 supports hot deployment which allows you to deploy new services at runtime. The top layer is dedicated to transport and data binding[2].

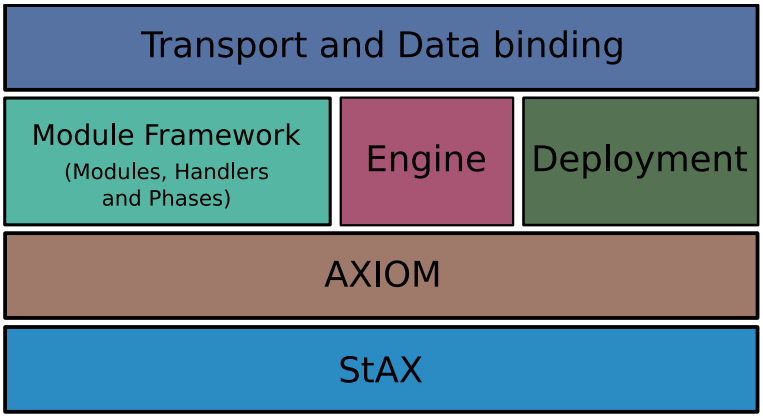

Figure 2.4: Axis2 Component Architecture

The Axis2 processing model is shown in Figure 2.5. Through its client API the user application can utilize a Web Service. The Axis2 client API supports *Handlers* for common processing needs. The *transportation sender* then can send the request as a SOAP envelope to the server side. Like the client side, Axis2 also have *Handlers* that the envelope can be processed through. This is where the WS-*[3] functionality is

[2]XML data binding refers to the process of representing the information in an XML document as an object in computer memory.

[3]WS-* is a common name on all the different WS standards.

implemented. Through a chain of processing handlers, different requirements can be enforced transparently for the user application and the Web Server business logic.

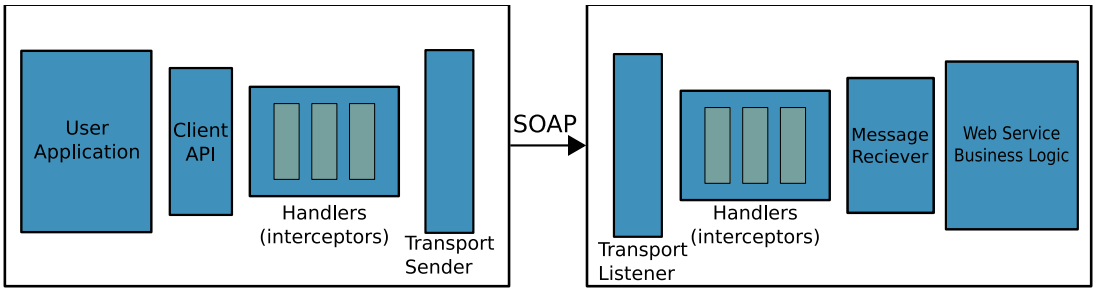

Figure 2.5: Axis2 Processing Model

2.3.2 XFire

XFire is a relatively new framework comparing to the Apache Axis project. It started in 2004 and one of its primary goals has been to create an embeddable and intuitive API. XFire uses the Spring Framework[4] internally which has a strong focus on *low coupling and high cohesion* [24] through the *Inversion of Control* pattern [25].

Some of the key features in XFire are [26]:

- WSDL 1.1 Support
- StAX - Fast API for XML Processing
- Pluggable bindings - POJOs, XMLBeans, JAXB 1.1, JAXB 2.0, and Castor support
- JSR 181 API (Annotations) [27] to configure services
- Container support: Spring, Pico, Plexus, and Loom

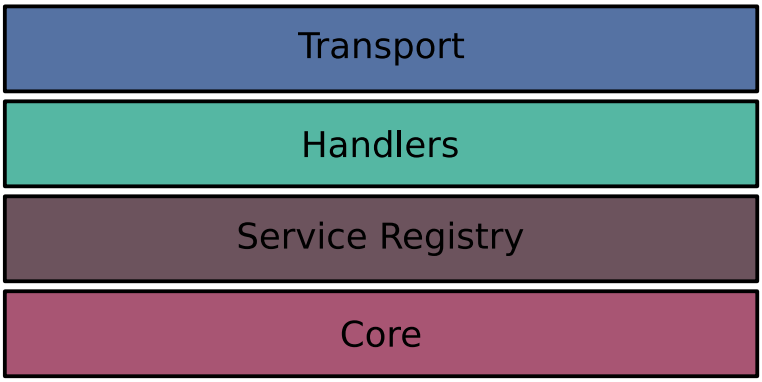

Figure 2.6: XFire Component Architecture

[4]Spring Framework: `http://springframework.org/`

The component architecture of XFire is shown in Figure 2.6, and has some of the same elements as Axis2. Through its central service registry it looks up services and then invokes the set of *Handlers* associated with that service. When the handlers are finished processing, the request is forwarded to a *Transport* component that handles transportation of the message to the client.

2.4 Security in Web Services

With the growing popularity of Web Services, comes the increased security threat against business systems. A Web Service creates additional entry points in an application. The more entry points an application has, the more entry points need to be secured by the development team.

Chapter 5 in [28] presents a comprehensive information security framework. It has six essential foundation elements. If any one of them is omitted, information security is deficient in protecting information owners. The foundation elements are:

- Availability
- Utility
- Integrity
- Authenticity
- Confidentiality
- Possession

Input validation mainly concerns the *Integrity* and *Authenticity* elements, according to [28]. Integrity is defined as completeness, wholeness and readability of information, and quality being unchanged from a previous state. Authenticity relate to validity, conformance and genuineness of information. Proper input validation is therefore one instrument to aid in keeping the integrity and authenticity of a system.

There are many strategies that can be used when validating user input. The two most common strategies are blacklisting and whitelisting [29]. Blacklisting filters data according to a list of forbidden elements. Whitelisting identifies allowed elements and removes everything else. Whitelisting is recommended because it is easier to identify a set of safe input, than it is to identify all possible bad input.

Chapter 3.2 in [29] presents a list of suggestions for good input validation practise. One of the suggestions is to automate the input validation process. Automatic input validation based on a set of policies will encourage developers to create more secure applications. Good automation procedures and an easy to use policy language is more appealing to a developer, than having to code input validation “manually”.

2.4.1 Security Threats

[18] divides Web Service into four different layers, with the purpose of grouping security threats. These layers are:

In transit: The SOAP envelope is created at a client and is (usually) transported over the Internet to a web server, this distance is called *In transit*. This is a very difficult layer to control and a lot of security issues regarding confidentiality, message integrity and trust are raised when handling the *in transit* layer.

Engine: The engine layer consists of the web server and/or web application server. These components receive and processes the SOAP envelope. They are usually vendor controlled and can have generic vulnerabilities. The end user normally does not have any control over this layer and have to trust their vendor regarding security threats in this layer.

Deployment: The deployment layer is controlled by the users deploying the Web Services. These threats are oriented around the deployment and configuration of the Web Service such as cryptography, authentication and customized errors.

User code: This area has the largest number of threats since this is where the envelope and its payload is finally processed and used. Sloppy coding practices open doors for attackers.

Chapter 4 in [18] has also listed the most common threats for each layer. They are repeated here in Table 2.2. The threats listed for layer 1 and 2 are out of the scope of this project, and is omitted since we primarily are focusing on input validation.

Web Services Layer	**Attacks and & Threats**
Layer 3 *Web Services Deployment*	1. Fault code leaks 2. Permissions & Access issues 3. Poor policies 4. Customized error leakage 5. Authentication and Certification
Layer 4 *Web Services User code*	1. Parameter tampering 2. WSDL probing 3. SQL/LDAP/XPATH/OS command injection 4. Virus/Spyware/Malware injection 5. Bruteforce 6. Data type mismatch 7. Content spoofing 8. Session tampering 9. Format string 10. Information leakage 11. Authorization

Table 2.2: Web Services Security Threats

The most prominent threats from an input validation point of view are:

- **Fault Code Leaks:** Sensitive information about the system revealed through fault codes as a response to malformed requests sent by an attacker.
- **Parameter Tampering:** An attacker modifies parameters to disclose sensitive information.
- **SQL/LDAP/XPATH/OS Command Injection:** This is an extension of Parameter Tampering, where the use of special meta characters can modify underlying operations in a subsystem used by the business logic.

- **Data Type Mismatch:** Developer coding errors or attackers deliberately changing data types of input, and possibly cause malfunctions in the business logic. End result is a possible exposure of sensitive internal information.

All of these threats can be drastically reduced, by validating all input before it reaches the Web Service business logic. Upon receiving malformed or invalid requests, a non-informative response should be generated [29].

2.4.2 Web Application Firewalls

[30] defines the term web application firewall: *An intermediary device, sitting between a web-client and a web server, analyzing OSI Layer-7 messages for violations in the programmed security policy. A web application firewall is used as a security device protecting the web server from attack.*

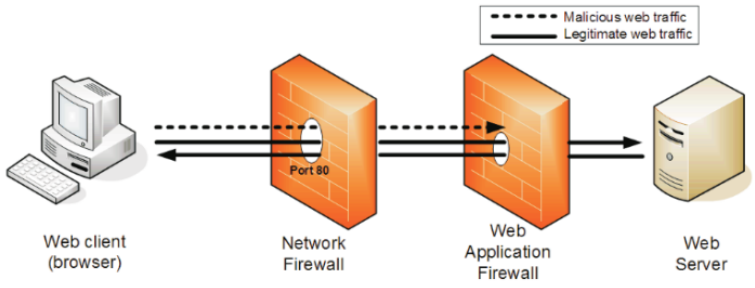

Figure 2.7: Web Application Firewall (from [31])

Figure 2.7 shows how the application firewall is added to the request chain when a client is sending a request to a web server. Web Services usually uses HTTP as the transport protocol. HTTP uses port 80 for all communication, and therefore the network firewall need to be open for all traffic on port 80. A normal packet firewall does not have the ability to inspect the content of a HTTP request. This is where the Web Application Firewall can be used. Since it is operating on the Layer 7 (Application Layer) of the OSI-model [32], it can inspect and block potentially dangerous input sent in HTTP requests.

Since the firewall is not a part of the applications it is protecting, it can be hard for a developer to test it. He might not even have the possibility to change the configuration or know how it is configured at all. Not knowing if the firewall is active or not, can be a huge security risk. Also the consequences if the firewall goes down, and the applications behind is left open can be devastating.

It does, however, provide centralized protection for all applications behind it. For preventing general attacks they can be great, but they have been shown to be cumbersome to tailor to a specific application [31]. A web application firewall is therefore not a *silver bullet* [33] to all input validation issues.

Chapter 3

Own Contribution

This chapter presents this master's thesis contribution work. With basis in the theory presented in Chapter 2, we have developed a Web Service input validation framework: WSPV - Web Service Payload Validator.

We will first discuss some issues regarding input validation with WSDL-documents. These issues form the background for the framework requirements. Then the implementation details of the project will be presented, followed by an example application that demonstrates an example use of the framework.

3.1 Input Validation with WSDL-documents

XML Schema types definitions in the WSDL-document, is the primary method of doing input validation in today's Web Services. Most SOAP frameworks support the basics of XML Schema types, but the basics are often not enough to be able to trust the data. Depending on the SOAP framework used on the client and server side, sent and received input does not necessarily comply with the constraints defined in the WSDL.

The two most common open source SOAP frameworks Axis2 [22] and XFire [26] does not come with "out of the box"[1] WSDL validation. With XML data binding extensions WSDL validation can be performed, but this process is different in every framework and may or may not be easy to activate.

Another issue with SOAP frameworks is their ability to accept custom WSDL schema types. Most frameworks provide automatic creation of WSDL documents based on the defined services. Some frameworks provide mechanisms that require few and simple steps for using custom schema types, others can be a bit more troublesome to configure.

The verboseness of WSDL documents is perhaps one of the key problems with WSDL validation. In Section 2.1.2 about XML Schemas, we showed in Listing 2.9 that a small type definition with only a few parameters, required a total of 38 lines of XML. Normally you would use some of the existing WSDL generator tools to create the basis for these schemas. This saves the developer for a lot of tedious and error prone work, but at the end most schemas require quite a bit of manual intervention to create a solid and restrictive schema. For example, most generators translate a Java *String* as an *xsd:string* type, even though it might represent a license plate number or a social security number that has more restrictions. In most cases a developer would therefore be required to manually edit the WSDL document, since it most likely is not restrictive enough from a security perspective.

To summarize, there are quite a few issues related to input validation via WSDL-documents. The most prominent issues are listed below.

- XML Schemas in itself are complex.
- Its complexity makes it a time consuming operation to manually write schema definitions.
- The support for XML Schema validation are limited and varying between each SOAP framework.
- WSDL documents are often published to third parties, and changes in the XML Schema types would then have to be republished to all involved parties.

[1] Out of the box features are those features that do not require any additional actions from the developer to be activated.

3.2 Requirements

In this project we have used eXtreme Programming (XP) as the development methodology, and Test Driven Development (TDD) as the development technique. The rationale for using XP and TDD is that the project is quite small, and when we started we only knew a couple of the features that we wanted. We knew that we most likely would find new requirements during the development phase, and therefore a development methodology with short cycles and continuous feedback would be great. Next we will give a short introduction to XP and TDD before the requirements themselves is presented.

eXtreme Programming (XP) is a relatively new software development methodology. It was created by Kent Beck in the middle of the 1990s, as an alternative to the linear waterfall model [34]. XP emphasizes customer involvement and short development cycles that result in early, concrete and continuous feedback [35]. XP uses user stories to gather requirements. User Stories are short descriptions of functionality visible to the user. "A user story is the smallest amount of information (a step) necessary to allow the customer to define a path through the system" [35].

Test Driven Development (TDD) is an agile software development technique [36] where you write the tests first and write the code afterwards. This means that the requirements are translated into tests and then the tests specify how the code should behave. The large amount of tests written also increases the chance of discovering errors during development.

The user stories were mainly acquired through talks with our supervisor Carl C. Christensen at BEKK Consulting AS and their experiences with security issues in Web Services. The overall goals with the framework was to satisfy the security issues discussed in Section 3.1.

3.2.1 User Stories

These are the user stories used as development requirements for the WSPV framework. All user stories are prefixed with US-X and numbered so they can be referred to later.

US-1 *Automatic validation of SOAP envelope payloads*

A SOAP request is received and the payload is automatically validated based on a set of policies.

US-2 *Configure validation policies*

The user can configure validation policies for each service at one central location. All services within the same application is configured from the same location.

US-3 *Validate single elements*

Each element in the payload can be configured with its own validation policy.

US-4 *Validate complex objects*

Complex objects consisting of other objects can be configured with its own validation policy.

US-5 *Standard validators*

The user can choose between a set of standard validator to validate common datatypes such as integer, double, string, date, email and URL.

US-6 *Create custom validators*

A developer can create custom validators for custom datatypes.

US-7 *No SOAP framework dependencies*

The core of the framework should not have any dependencies to any SOAP framework.

US-8 *XFire support*

The framework should support the XFire SOAP framework through their handler interface.

US-9 *Require validation of all elements*

A developer should be able to force validation of all elements present in an envelope, and a fault should be returned if the envelope contains elements without any validation policies.

US-10 *Apache Axis2 support*

The framework should support the Apache Axis2 SOAP framework through their handler interface.

3.3 Implementation

This section describes the implementation of the WSPV framework. Regular class diagrams and other appropriate UML diagrams are used to present the details of the framework.

When we designed the architecture we tried to use common known architectural patterns to help achieve the wanted qualities of the framework. Especially US-6, US-7, US-8 and US-10 have influenced the choice of *Low coupling* [24, 16.8] and *High cohesion* [24, 16.9] as architectural patterns.

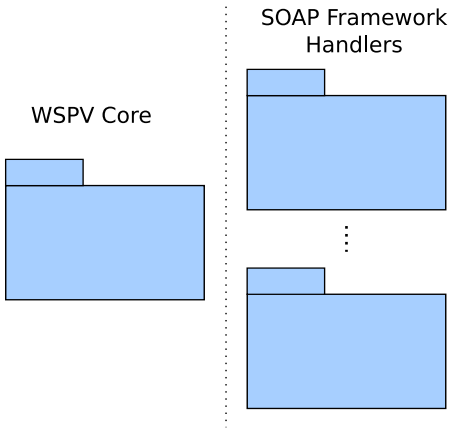

Figure 3.1: WS Payload Validator - Parts Overview

The framework has been split into two separate parts, core and handlers. This is illustrated in Figure 3.1. To support US-7, US-8 and US-10, we need to keep the core separated in order to provide clean interfaces for other SOAP frameworks. For a SOAP framework to utilize WSPV, a handler must be created to hook into the core of the framework. By separating it like this we can support multiple SOAP frameworks just by developing a new handler for each SOAP framework and keep the core consistent.

3.3.1 WS Payload Validator - Core

The core of the WSPV framework is simple but powerful. Figure 3.2 shows the core components of the framework in a UML class diagram.

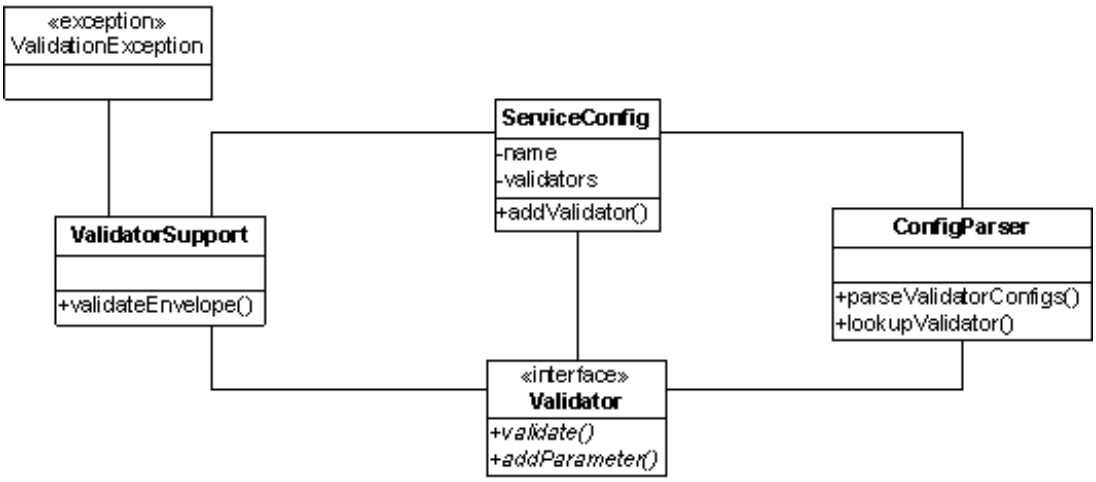

Figure 3.2: WS Payload Validator Core - UML Class Diagram

We will give a brief introduction to the purpose and functionality of the different components in the following sub sections.

no.fjas.wspv.ServiceConfig (class)

The *ServiceConfig* class holds the policy configuration for all Web Services within the same application. Based on the policies that have been configured, the correct and fully configured validator can be extracted from the ServiceConfig and validate the XML element.

The source code for the *ServiceConfig* class is listed in appendix B.1.

no.fjas.wspv.Validator (interface)

The *Validator* interface must be implemented by all validators. A validator can verify the content of an XML element and has two methods, *validate()* and *addParameter()*. The *validate()* method is used to perform validation on an XML element and the *addParameter()* method configures the validator with extra parameters to tailor it to a specific element. E.g. an integer validator may have a minimum and maximum value, these values are configured in the policy file and injected into the validator at application startup through the *addParameter* method.

The source code for the *Validator* interface is listed in appendix B.2.

no.fjas.wspv.ConfigParser (class)

The ConfigParser helps parsing the policy configuration file and builds ServiceConfig objects. The ServiceConfig are then used for looking up Web Service endpoints[2], and returns the appropriate validator during the validation procedures.

The source code for the *ConfigParser* class is listed in appendix B.3.

no.fjas.wspv.ValidatorSupport (class)

ValidatorSupport is, as the name implies, a helper class for performing validation of SOAP envelopes. It can read SOAP envelopes and perform validation on all elements based on the current ServiceConfig. This class should be called by a SOAP framework handler when a new SOAP envelope request is received. If the validation policy fails, it will throw a ValidationException and abort the current request.

The source code for the *ValidatorSupport* class is listed in appendix B.4.

3.3.2 WS Payload Validator - Policy Configuration

Policy configuration in WSPV is based on XML. The configuration syntax is very minimalistic and the a DTD for the policy configuration is attached in appendix A.

A sample configuration file is shown in Listing 3.1. The configuration file is oriented around available services and their operations. A *service* can have multiple operations available. Each *operation* can take one or more parts as arguments. With each *part* there is an attribute called *type*. The *type* attribute tells WSPV which validator class to use when validation is performed. In addition to the *type* attribute, a *part* element can take several parameters to customize a standard validator.

```
<validation>
  <service name="BookService">
    <operation name="findBook">
      <part name="isbn" type="regex">
        <param name="expression">
          (?=.{13}$)\d{1,5}([- ])\d{1,7}\1\d{1,6}\1(\d|X)$
        </param>
      </part>
    </operation>
  </service>
</validation
```

Listing 3.1: WS Payload Validator - Configuration Policy Example

In Listing 3.1 we configure the *findBook* operation of the *BookService*. This operation takes a single argument, *isbn*, which is of the type *regex*. This is a regular expression validator that accepts a regular expression as a parameter.

[2] An endpoint indicates a specific location for accessing a Web Service using a specific protocol and data format. [19]

With the WSPV framework we have developed a set of standard validators. These validators can be used to perform validation of the most basic standard data types. All validators must implement the *Validator* interface introduced in Section 3.3.1. The standard validators provided with WSPV are listed in Table 3.3.2.

Class name	**Type name**	**Validates**
DateRangeValidator	date	Dates
DoubleRangeValidator	double	Double numbers
EmailValidator	email	Email addresses
EnumValidator	enum	Enumeration values
IntRangeValidator	int	Integer numbers
MultiPartValidator	multipart	Objects
RegexValidator	regex	Regular expressions
StringLengthValidator	string	Strings
URLValidator	url	HTTP URLs

Table 3.1: WS Payload Validator - Provided Standard Validators

3.3.3 WS Payload Validator - XFire Handler

To demonstrate the use of WSPV, we need a connector to another SOAP framework. We decided to first start of with the Codehaus XFire framework [26] and built a customized handler that intercepts SOAP envelopes and uses WSPV for payload validation.

Figure 3.3 shows the UML sequence diagram of how the XFire handler communicates with the WS Payload Validator.

The handler itself is really simple. By hooking into XFire's chain of handlers, we are able to retrieve the SOAP envelope from the client, fetch the payload (SOAP body) and run validation against the policies defined in WSPV. All this is done transparently for both the client and the developer of the business logic. In case of validation failure a SOAP Fault will be generated and returned to the client. Upon successful validation, the handler will not interfere, and the request will be processed normally.

The source code for the XFire handler is included in appendix C.

3.3.4 WS Payload Validator - Axis2 Handler

We have also developed a handler for the Apache Axis2 [22] SOAP framework. This utilizes WSPV in the same way as XFire, but since the context and API is a bit different from XFire we had to develop a customized handler for Axis2. No sequence diagram has been created as it would be a duplicate of Figure 3.3.

The source code for the Axis2 handler is included in appendix D.

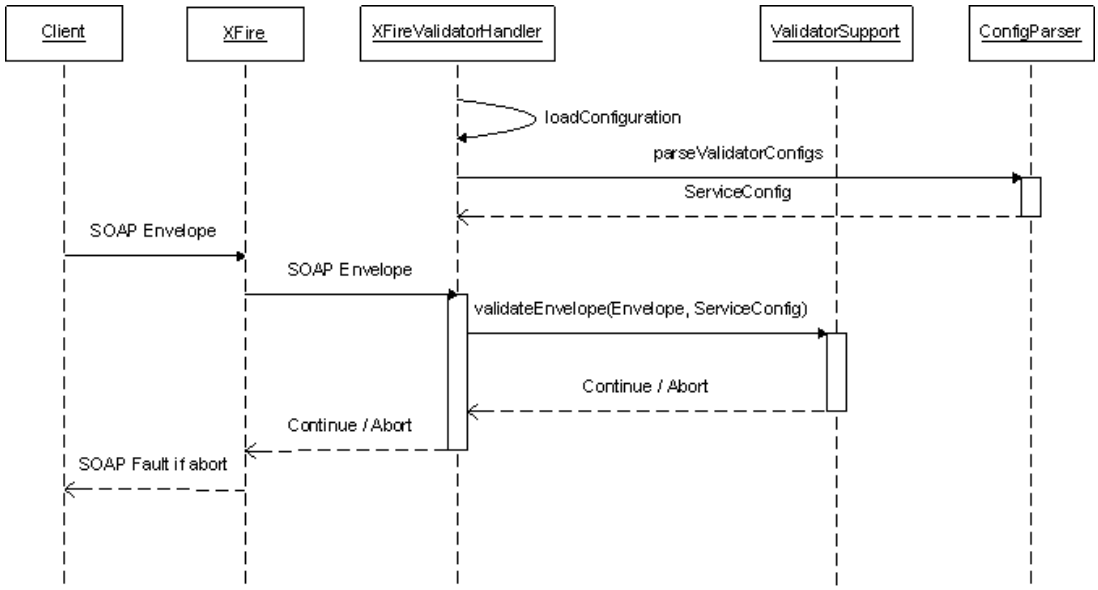

Figure 3.3: XFire Handler - UML Sequence Diagram

3.4 Example Application - Book Service

One of the goals of this project is to demonstrate the use of the framework. We have fulfilled this goal by creating an example application, a book service.

Our example is a Java web application that holds information about a collection of books. We wish to expose a set of services, so the rest of the world can make use of our book collection. We want to make the following operations available through a Web Service:

- getBooks - retrieves a list of all books
- findBook - lookup a book by isbn number
- getBook - lookup a book by id number
- findBooksByMaxPrice - retrieves a list of books satisfying a given price range
- addBook - adds a book to the collection

3.4.1 Web Service Implementation

The book service is exposed through the *BookService* interface shown in listing 3.2.

```
public interface BookService {

    public Book[] getBooks();

    public Book findBook(String isbn) throws BookException;

    public Book getBook(int id) throws BookException;

    public Book[] findBooksByMaxPrice(double price);

    public void addBook(Book book);
}
```

Listing 3.2: Example Application - BookService.java

For the *BookService* interface to be usable we need an implementation class. Listing 3.3 shows the *BookServiceImpl* class. This is a simplified implementation class and there are no database lookups or any other advanced data management here. In an ordinary application this class would often forward requests to a service layer [37].

```
public class BookServiceImpl implements BookService {
        private Map<String, Book> bookList = new HashMap<String, Book>();

        public BookServiceImpl() {
                Book book1 = new Book();
                book1.setId(1);
                book1.setAuthor("Dan Diephouse");
                book1.setTitle("Using XFire");
                book1.setIsbn("12-3456-789-0");
                book1.setPrice(49.99);
```

```
                Book book2 = new Book();
                book2.setId(2);
                book2.setAuthor("J. K. Rowling");
                book2.setTitle("Harry Potter and the Deathly Hallows ");
                book2.setIsbn("0545010225");
                book2.setPrice(34.99);

                bookList.put(book1.getIsbn(), book1);
                bookList.put(book2.getIsbn(), book2);
        }

        public Book[] getBooks() {
                Collection<Book> books = bookList.values();
                return (Book[]) books.toArray(new Book[books.size()]);
        }

        public Book findBook(String isbn) throws BookException {
                Book result = bookList.get(isbn);
                if (result != null)
                        return result;
                else
                        throw new BookException("Book does not exists",
                                        new BookExceptionDetail("NOT_EXIST", "Can't
                                             find book"));
        }

        public Book getBook(int id) throws BookException {
                Iterator<Book> iter = bookList.values().iterator();
                while (iter.hasNext()) {
                        Book book = iter.next();
                        if (book.getId() == id)
                                return book;
                }

                throw new BookException("Book does not exists",
                                new BookExceptionDetail("NOT_EXIST", "Can't find book
                                     "));
        }

        public Book[] findBooksByMaxPrice(double price) {
                Collection<Book> bookCollection = bookList.values();
                Iterator<Book> iter = bookCollection.iterator();
                Book[] books = new Book[bookCollection.size()];
                int counter = 0;
                while (iter.hasNext()) {
                        Book book = iter.next();
                        if (book.getPrice() >= price)
                                books[counter++] = book;
                }
                return books;
        }

        public void addBook(Book book) {
                bookList.put(book.getIsbn(), book);
        }

}
```

Listing 3.3: Example Application - BookServiceImpl.java

The Web Service is created using the XFire SOAP framework [26]. To expose a Web Service we need to configure the SOAP framework, and with XFire this is done

through an XML configuration file. The configuration file must reside in the directory *META-INF/xfire* and named *services.xml*. Configuration for our book service example is shown in Listing 3.4.

```
<beans xmlns="http://xfire.codehaus.org/config/1.0">
  <service>
    <name>BookService</name>
    <namespace>http://wspv.fjas.no/BookService</namespace>
    <serviceClass>no.fjas.wspv.examples.xfire.BookService</serviceClass>
    <implementationClass>
      no.fjas.wspv.examples.xfire.BookServiceImpl
    </implementationClass>
  </service>
</beans>
```

Listing 3.4: Example Application - XFire configuration, services.xml

We have now completed our Web Service. We can start the service and clients can communicate with our book service. A *findBook* request and response to the presented service would now look exactly like those referred to in 2.2.1, Listing 2.7 and 2.8.

3.4.2 Adding Input Validation

We now want to add input validation to our Web Service. Adding input validation requires two steps. Create a connection to our WS Payload Validator framework and create validation policies.

Connecting the book service with WS Payload Validator is done with the XFireValidatorHandler presented in Section 3.3.3. This requires a few additional lines to the XFire configuration file *services.xml*. Listing 3.5 shows the complete configuration including line 13-18 required for connecting the book service to the validation handler.

```
1  <beans xmlns="http://xfire.codehaus.org/config/1.0">
2    <service>
3      <name>BookService</name>
4      <namespace>
5        http://wspv.fjas.no/BookService
6      </namespace>
7      <serviceClass>
8        no.fjas.wspv.examples.xfire.BookService
9      </serviceClass>
10     <implementationClass>
11       no.fjas.wspv.examples.xfire.BookServiceImpl
12     </implementationClass>
13     <inHandlers>
14      <handler
15        handlerClass="org.codehaus.xfire.util.dom.DOMInHandler"/>
16      <handler
17        handlerClass="no.fjas.wspv.xfire.WSPVHandler"/>
18     </inHandlers>
19   </service>
20 </beans>
```

Listing 3.5: Example Application - XFire configuration with validation, services.xml

Next step is to add validation policies. Recall the *BookService* interface and the available methods listed in Listing 3.2. The method *getBooks* does not need any validation policy since it does not accept any arguments. The rest, however, need to be taken care of. We have listed a policy proposal in Table 3.2.

Method	**Arguments**	**Policy**
findBook	isbn (string)	regular expression
getBook	id (int)	int > 0
findBookByMaxPrice	price (double)	double > 0.0
addBook	*book (object)*	
	author (string)	max length = 255
	id (int)	int > 0
	isbn (string)	regular expression
	price (double)	double > 0.0
	title (string)	max length = 255

Table 3.2: Example Application - Validation Policy proposal

We will explain these policies briefly. The isbn number is quite complex and is built up by a sequence of numbers and dashes. We will validate this using a regular expression [3]. We also want all of the integer and double values to be positive, and the strings must not exceed a max length of 255.

We need to translate these policies into a understandable form for the WSPV framework. All the policies must be defined in the *validation.xml* file in the *META-INF/wspv* directory. The policy configuration for the book service is listed in 3.6.

```
<validation>
  <service name="BookService">
    <operation name="findBook">
      <part name="isbn" type="regex">
        <param name="expression">
          (?=.{13}$)\d{1,5}([- ])\d{1,7}\1\d{1,6}\1(\d|X)$
        </param>
      </part>
    </operation>
    <operation name="getBook">
      <part name="in0" type="int">
        <param name="minExclusive">0</param>
      </part>
    </operation>
    <operation name="findBooksByMaxPrice">
      <part name="in0" type="double">
        <param name="minExclusive">0</param>
      </part>
    </operation>
    <operation name="addBook">
```

[3] More about regular expressions: `http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html`

```
      <part name="in0" type="multipart">
        <part name="author" type="string">
          <param name="maxLength">255</param>
        </part>
        <part name="id" type="int">
          <param name="minExclusive">0</param>
        </part>
        <part name="isbn" type="regex">
          <param name="expression">
            (?=.{13}$)\d{1,5}([- ])\d{1,7}\1\d{1,6}\1(\d|X)$
          </param>
        </part>
        <part name="price" type="double">
          <param name="minExclusive">0.0</param>
        </part>
        <part name="title" type="string">
          <param name="maxLength">255</param>
        </part>
      </part>
    </operation>
  </service>
</validation>
```

Listing 3.6: Example Application - WS Payload Validator configuration

This was all that was needed to secure this Web Service. The WSPV Handler will now be invoked automatically each time a *BookService* operation is request by a client. If the client request data does not conform with the given policy, the request is aborted and a SOAP fault will be returned to the client.

Chapter 4

Evaluation and Conlusion

This chapter contains a discussion where we focus on what we have accomplished during this project. We will evaluate the Web Service Payload Validator framework we have implemented, with respect to the specified requirements and goals of the project.

Further on we will present the conclusion of the project and summarize our achievements. The last section covers further work and suggestions for a possible follow up project.

4.1 Discussion and Evaluation

The main achievement in this project is the creation of an input validation framework for Web Services. This was also the primary goal of the project. Through the literature study in Chapter 2, we became aware of several issues with today's solutions on performing proper input validation in Web Services. Most of the issues originated from the complexity in performing input validation (see Section 3.1). With the WSPV framework we have tried to reduce the complexity, and in Section 3.2.1 we specified a set of requirements for the framework to achieve this.

Through the development of the WSPV framework we have fulfilled all the requirements set in Section 3.2.1. With all of the requirements fulfilled, we are of the opinion that the solution has provided a treatment to the issues listed in Section 3.1.

A developer can now perform input validation of SOAP envelope payloads automatically, based on a set of policies, when using the WSPV framework. This is all done transparently, both for the client and the business logic on the server side. These two arguments are probably the most important ones. An automatic, simple and unobtrusive validation framework, with reusable policies and validators, are easier to use than WSDL-documents. This is shown through the example application that required a total of four extra XML lines in the XFire service configuration and eleven lines of XML in the policy configuration, to protect a single operation with one parameter. In Section 3.1 we showed that the WSDL-document requires a total of 38 lines of type definitions for that same single operation. In addition you have to enable validation of the WSDL-document, which can be a difficult task, depending on which SOAP framework you use.

Other advantages achieved by using the WSPV framework is the independence of SOAP frameworks. With WSPV you have the possibility to change SOAP framework without having to change the policies. The only thing that has to be changed is which connection handler you use to connect to the framework. This independence also opens up a future possibility for having a central base of policies, and use the same installation of the framework with different SOAP frameworks.

To support our beliefs about the mentioned advantages of the framework, we could have performed an empricical study to see if developers think it is easier to use than WSDL-documents. The time limit on the project did unfortunately not allow this. We have, however, made contact with the people behind the XFire SOAP framework and will try to get the framework introduced to the XFire developer community.

During the development of the framework we came across several situations that made us redefine our requirements. The need for separation between the WSPV core and SOAP frameworks, was revealed when we had to choose which SOAP framework to use for the example application. Many popular frameworks to choose from made it clear that the framework needed to be as generic as possible, so we could support multiple SOAP frameworks.

The default settings in the WSPV framework are also important. Even though they are not specified formally in the requirements, there are several settings in the framework

that provide what some people may call strict settings. An example is null (empty) values, which are not allowed unless people state explicitly in the policy that this element is not required. Unless this is stated, the WSPV framework will abort the current request upon recognition of a null value.

[38] acknowledges that the default setting in a piece of software is very important. Strong defaults protect novice users from adverse consequences. [38] also states several examples from software and security settings, where the vast majority of users never changed any settings. This supports our belief of having automatic tools with strong defaults that will increase the default security in a Web Service.

4.2 Conclusion

Secure Web Services is important since they often provide end points directly into the business logic of an application. Because Web Services is a relatively new technology, many web applications have been extended with Web Service functionality to fulfill change in demands. This may, however, cause security issues since this functionality was not planned in the beginning. Security mechanisms may be focused around the web application's user interface, and not the business logic layer. When directly exposing business logic with Web Services, we need security mechanisms to protect ourselves against malicious input.

During this project we have created an input validation framework for Web Services, Web Service Payload Validator (WSPV). Simple policy configuration, reusable policies and support for several SOAP frameworks, makes this framework a valueable addition to the toolbox of a Web Service developer. By not relying on the WSDL-documents, we have created an alternative for those developers not using or wanting to use XML Schemas for input validation. For those who already use XML Schemas, this framework can be utilized as an additional tool to increase *Defence-in-depth* [39].

With the example application we developed, we have demonstrated how input validation security can be added transparently after the Web Service itself has been set in production. A common factor for encouraging developers to create secure applications is ease of use. Complex and error prone procedures does not encourage developers to implement security. The unobtrusive and transparent nature of the WSPV framework, makes it easy for developers to increase security.

4.3 Further Work

This section contains a list of features and suggestions that could be implemented to improve the Web Service Payload Validator framework. These are:

- Make the framework Open Source. We have started the work on making the software Open Source, and we need to continue this work and promote it in the Web Service community.
- Expand the set of default validators. Currently only a small set of validators is provided, supporting only the most common data types.
- Implement support for more SOAP frameworks.
- The framework could be restructured to become a central validation service and operate on a range of applications instead of just one. By having the policies and the validation routine running on a central server and only distribute a connector to each application, this enhance reuse of policies and reduce maintenance in large production environments.

The first three suggestions are not big or complex work. By making the software Open Source, we can get valuable feedback from the community on those features we already have and perhaps suggestions on new features. The last suggestion about centralized validation, would move this project into a more of a web service firewall. This would require quite a bit of work. The entire project will have to be restructured when it comes to how it is connected to each web application and how the configuration is stored. It is, however, a very interesting project that adds aspects of infrastructure and performance to the project.

Bibliography

[1] Symantec Corporation. Symantec internet security threat report - volume xi: March 2007, March 2007. http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_internet_security_threat_report_xi_03_2007.en-us.pdf (last accessed 10.05.07).

[2] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation, http://www.w3.org/TR/soap12-part1/, June 2003.

[3] Rune Frøysa Åsprang and Lars Arne Brekken. Adding security to web services. Master's thesis, Norwegian University of Science and Technology (NTNU), June 2006.

[4] Steve Graham, Doug Davis, Simeon Simeonov, Glen Daniels, Peter Brittenham, Yuichi Nakamura, Paul Fremantle, Dieter Koenig, and Claudia Zentner. *Building Web Services with Java : Making Sense of XML, SOAP, WSDL, and UDDI (2nd Edition) (Developer's Library)*. Sams, June 2004.

[5] Hiroshi Maruyama, Kent Tamura, and Naohiko Uramoto. *XML and Java: developing Web applications*. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 2002.

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Third Edition). W3C Recommendation, http://www.w3.org/TR/REC-xml, February 2004.

[7] T. Bray, D. Hollander, and A. Layman. Namespaces in XML. W3C Recommendation, http://www.w3.org/TR/REC-xml-names, January 1999.

[8] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures. W3C Recommendation, http://www.w3.org/TR/xmlschema-1, May 2001.

[9] David C. Fallside and Priscilla Walmsley. XML Schema Part 0: Primer Second Edition. W3C Recommendation, http://www.w3.org/TR/xmlschema-0/, May 2001.

[10] Geert Jan Bex, Frank Neven, and Jan Van den Bussche. Dtds versus xml schema: a practical study. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 79–84, New York, NY, USA, 2004. ACM Press.

[11] Geert Jan Bex, Wim Martens, Frank Neven, and Thomas Schwentick. Expressiveness of xsds: from practice to theory, there and back again. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 712–721, New York, NY, USA, 2005. ACM Press.

[12] David Booth, Christopher Ferris, Mike Champion, David Orchard, Francis McCabe, Hugo Haas, and Eric Newcomer. Web services architecture, February 2004. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[13] Martin Gudgin, Marc Hadley, and Tony Rogers. Web services addressing 1.0 - core, May 2006. http://www.w3.org/TR/2006/REC-ws-addr-core-20060509.

[14] Prasad Yendluri, Toufic Boubez, David Orchard, Asir S. Vedamuthu, Ümit Yalçınalp, Maryann Hondo, and Frederick Hirsch. Web services policy 1.5 - framework, March 2007. http://www.w3.org/TR/2007/CR-ws-policy-20070330.

[15] Anthony Nadalin, Chris Kaler, Ronald Monzillo, and Phillip Hallam-Baker. Web services security: Soap message security 1.1, February 2006. http://docs.oasis-open.org/wss/v1.1/.

[16] Anthony Nadalin, Marc Goodner, Martin Gudgin, Abbie Barbir, and Hans Granqvist. Ws-securitypolicy 1.2, March 2007. http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702.

[17] Kazunori Iwasa, Jacques Durand, Tom Rutt, Mark Peel, Sunil Kunisetty, and Doug Bunting. Web services reliable messaging tc, ws-reliability 1.1, November 2004. http://docs.oasis-open.org/wsrm/ws-reliability/v1.1.

[18] Shreeraj Shah. *Hacking Web Services*. Charles River Media, Inc., Rockland, MA, USA, 2006.

[19] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1, March 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[20] Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. Uddi version 3.0.2, October 2004. http://uddi.org/pubs/uddi_v3.htm.

[21] The Apache Software Foundation. http://ws.apache.org/soap. Apache SOAP.

[22] Apache Software Foundation. http://ws.apache.org/axis2. Apache Axis2.

[23] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Inter. Tech.*, 2(2):115–150, 2002.

[24] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[26] Envoi Solutions LLC. http://xfire.codehaus.org. Codehaus XFire.

[27] Stuart Edmondston and Brian Zotter. Jsr 181: Web services metadata for the java platform, February 2005. http://jcp.org/en/jsr/detail?id=181.

[28] Seymour Bosworth and Michel Kabay. *Computer Security Handbook*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[29] Sverre H. Huseby. *Innocent Code - a security wake-up call for web programmers*. John Wiley & Sons ltd., 2004.

[30] Ivan Ristic, Robert Auger, Cesar Currudo, Jeremiah Grossman, Dennis Groves, Sverre H. Huseby, Aaron C. Newman, and Ray Pompon. Web security glossary. Retrieved May 2007, from: `http://www.cert.org/stats/`.

[31] Lieven Desmet, Frank Piessens, Wouter Joosen, and Pierre Verbaeten. Bridging the gap between web application firewalls and web applications. In *FMSE '06: Proceedings of the fourth ACM workshop on Formal methods in security*, pages 67–77, New York, NY, USA, 2006. ACM Press.

[32] H. Zimmermann. Osi reference model - the iso model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

[33] Jr. Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1978.

[34] Igor Hawryszkiewycz. *Systems Analysis and Design*. Prentice Hall, 2001.

[35] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999.

[36] Jeff Langr. *Agile Java, Crafting Code with Test-Driven Development*. Prentice Hall, 2004.

[37] Martin Fowler, David Rice, and Matthew Foemmel. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002.

[38] Rajiv C. Shah and Jay P. Kesan. Policy through software defaults. In *dg.o '06: Proceedings of the 2006 international conference on Digital government research*, pages 265–272, New York, NY, USA, 2006. ACM Press.

[39] C.L. Smith and M. Robinson. The understanding of security technology and its applications. In *Proceedings IEEE 33rd Annual 1999 International Carnahan Conference on Security Technology*, pages 26–37. IEEE Xplore, 1999.

Appendices

Appendix A

Policy Configuration DTD

Listing A.1 shows the Document Type Definition (DTD) that defines the structure of WSPV policy configuration file.

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

<!ELEMENT validation (service+)>

<!ELEMENT service (operation+)>
<!ATTLIST service
        name CDATA #REQUIRED
        validateEverything (true|false) "false"
>

<!ELEMENT operation (part+)>
<!ATTLIST operation
        name CDATA #REQUIRED
>

<!ELEMENT part (param*,part*)>
<!ATTLIST part
        name CDATA #REQUIRED
        type CDATA #REQUIRED
        required (true|false) "true"
>

<!ELEMENT param (#PCDATA)>
<!ATTLIST param
        name CDATA #REQUIRED
>
```

Listing A.1: WSPV Policy Configuration DTD

Appendix B

Source code - WSPV Core

This appendix contains the source code for the core of the WSPV framework. Only the main parts are listed here. The complete source code has been submitted as a digital appendix.

B.1 Source Code - *no.fjas.wspv.ServiceConfig*

Listing B.1 shows the source code for the *no.fjas.wspv.ServiceConfig* class.

```
package no.fjas.wspv;

import java.util.HashMap;
import java.util.Map;

/**
 * Holds the the configuration for a single Service. Consists of a
 * {@link java.util.Map} where the key is built up by
 * ServiceName/OperationName/ParameterName. The value is a Validator object that
 * has been initialized with the values from the policy configuration file.
 * <br><br>
 * E.g. BookService/findBook/isbn will return a RegexValidator with a regular
 * expression for validating ISBN-numbers.
 *
 * @author <a href="mailto:henning _at_ fjas.no">Henning Jensen</a>
 */
public class ServiceConfig {
  private String name;
  private boolean validateEverything = false;

  /** The Map containing all validators; key is Service/Operation/Parameter */
  private Map<String, Validator> validators = new HashMap<String, Validator>();

  public boolean isValidateEverything() {
    return validateEverything;
  }

  public void setValidateEverything(boolean mustValidateEverything) {
```

```
    this.validateEverything = mustValidateEverything;
  }

  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  public Map<String, Validator> getValidators() {
    return validators;
  }

  /**
   * Adds a validator to the configuration object.
   * @param key The key must be on the form: Service/Operation/Parameter.
   * @param validator The validator to validate the specified parameter.
   */
  public void addValidator(String key, Validator validator) {
    validators.put(key, validator);
  }
}
```

Listing B.1: Source Code - no.fjas.wspv.ServiceConfig

B.2 Source Code - *no.fjas.wspv.Validator*

Listing B.2 shows the source code for the *no.fjas.wspv.Validator* interface.

```
package no.fjas.wspv;

import org.w3c.dom.Element;

/**
 * Validates an SOAP-envelope. The {@link validate} method is called upon
 * invocation of the validation from a SOAP framework handler. Through the
 * {@link addParameter} method, parameters can adjust the behaviour of the
 * validator.
 *
 * @author <a href="mailto:henning _at_ fjas.no">Henning Jensen</a>
 * @version $Id: Validator.java 55 2007-03-14 10:25:48Z henjens $
 */
public interface Validator {

  /**
   * Performs validation on the given {@link org.w3c.dom.Element}. Fetches
   * node name and text content from the element.
   *
   * @param element the element to validate
   * @throws ValidationException when validation fails
   */
  void validate(Element element) throws ValidationException;

  /**
   * Adds parameters to configure a validator
   *
```

```
   * @param key the name of the parameter to add
   * @param value the value of the parameter to add
   * @throws ValidatorException
   *             on invalid parameters. Unknown parameters does not throw an
   *             exception only a log warning.
   */
  void addParameter(String key, String value) throws ValidatorException;
```

Listing B.2: Source Code - no.fjas.wspv.Validator

B.3 Source Code - *no.fjas.wspv.ConfigParser*

Listing B.3 shows the source code for the *no.fjas.wspv.ConfigParser* class.

```
package no.fjas.wspv;

import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

import no.fjas.wspv.util.ClassLoaderUtil;
import no.fjas.wspv.util.DomHelper;
import no.fjas.wspv.validators.MultiPartValidator;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;

/**
 * Loads web service validation configuration and validator configurations.
 *
 * @author <a href="mailto:henning _at_ fjas.no">Henning Jensen</a>
 */
public class ConfigParser {
  private static Log log = LogFactory.getLog(ConfigParser.class);

  private static ReadWriteLock lock = new ReentrantReadWriteLock();

  private static Map<String, String> availableValidators = new HashMap<String, String
      >();

  /**
   * Parses the validation configuration xml and builds up a
   * {@link java.util.Map Map} that holds all validation rules for all
   * services.
   *
   * @param inStream
   *            the stream holding the configuration file
   * @param resourceName
   *            the name of the configuration file (for debugging purposes
   *            only)
   * @return a Map with
```

```
     *         {@link no.fjas.wspv.ServiceConfig ServiceConfig}s
     *         mapped by service name.
     */
    public static Map<String, ServiceConfig> parseValidatorConfigs(
        InputStream inStream, String resourceName) {
      Map<String, ServiceConfig> configs = new HashMap<String, ServiceConfig>();
      Document doc = null;

      InputSource in = new InputSource(inStream);
      in.setSystemId(resourceName);

      doc = DomHelper.parse(in, null);

      if (doc != null) {
        NodeList serviceNodes = doc.getElementsByTagName("service");

        for (int i = 0; i < serviceNodes.getLength(); i++) {
          Element serviceElement = (Element) serviceNodes.item(i);

          ServiceConfig serviceConfig = readServiceConfig(serviceElement);
          configs.put(serviceConfig.getName(), serviceConfig);
        }

        return configs;
      } else {
        return null;
      }
    }

    /**
     * XML helper method reading in one single &lt;service&gt; element.
     *
     * @param serviceElement
     *            the xml element containing a service validation definition
     * @return a ServiceConfig with validators for each operation and part.
     */
    private static ServiceConfig readServiceConfig(Element serviceElement) {
      ServiceConfig config = new ServiceConfig();

      String serviceName = serviceElement.getAttribute("name");
      config.setName(serviceName);

      String serviceValidateEverything = serviceElement.getAttribute("
          validateEverything");
      config.setValidateEverything(Boolean.parseBoolean(serviceValidateEverything));

      NodeList operations = serviceElement.getElementsByTagName("operation");
      readOperationElements(config, serviceName + "/", operations);

      return config;
    }

    /**
     * Reads in all operation elements from a service definition.
     *
     * @param config
     *            the configuration object to load into
     * @param path
     *            the name of the current path to identify a unique validator
     * @param operations
     *            the operation elements to load
     */
    private static void readOperationElements(ServiceConfig config,
        String path, NodeList operations) {
```

```
    for (int i = 0; i < operations.getLength(); i++) {
      Element operationElement = (Element) operations.item(i);

      String operationName = operationElement.getAttribute("name");

      NodeList parts = operationElement.getChildNodes();

      StringBuilder newPath = new StringBuilder(path);
      newPath.append(operationName);
      newPath.append("/");

      readPartElements(config, newPath.toString(), parts);
    }
  }

  /**
   * Read part elements. Part elements can consists of params or another part
   * element (see validation.dtd)
   *
   * @param config
   *            config the configuration object to load into
   * @param path
   *            the name of the current path to identify a unique validator
   * @param parts
   *            the part elements to load
   */
  private static void readPartElements(ServiceConfig config, String path,
      NodeList parts) {
    for (int i = 0; i < parts.getLength(); i++) {
      Element partElement = (Element) parts.item(i);

      StringBuilder fullPath = new StringBuilder(path);
      fullPath.append(partElement.getAttribute("name"));

      String validatorType = partElement.getAttribute("type");
      if ((validatorType == null || validatorType.equals(""))
          && log.isDebugEnabled())
        log.debug("Element validator type is null for node named "
            + partElement.getNodeName() + "|"
            + partElement.getAttribute("name") + "| ->"
            + partElement);
      Validator validator = lookupValidator(validatorType);

      if (validatorType.equals("multipart")) {
        if (validator instanceof MultiPartValidator) {
          MultiPartValidator multipartValidator = (MultiPartValidator) validator;
          readMultiPartElement(partElement, multipartValidator);
        } else {
          // if this occurs, there has been a major programming error
          throw new ValidatorException(
              "Validator is not a multipart validator!");
        }
      } else {
        readPartParameters(partElement, validator);
      }

      config.addValidator(fullPath.toString(), validator);
    }
  }

  /**
   * Reads a part element that contains other part elements into a
   * {@link no.fjas.wspv.validators.MultiPartValidator MultiPartValidator}.
   *
```

```
   * @param multiPartElement
   *            the part element to read
   * @param parentValidator
   *            the validator to load configuration into
   */
  private static void readMultiPartElement(Element multiPartElement,
      MultiPartValidator parentValidator) {
    if (log.isDebugEnabled())
      log.debug("fetching part elements from element |" + multiPartElement.
          getAttribute("name") + "|");

    NodeList children = multiPartElement.getChildNodes();

    for (int i = 0; i < children.getLength(); i++) {
      log.debug("part element list has size: " + children.getLength());
      log.debug("i is " + i);
      Element child = (Element) children.item(i);
      String childName = child.getAttribute("name");
      String validatorType = child.getAttribute("type");
      if (log.isDebugEnabled())
        log.debug("childname is " + childName + ", type is " + validatorType);

      if ((validatorType == null || validatorType.equals(""))
          && log.isDebugEnabled())
        log.debug("Element validator type is null: " + child);

      Validator validator = lookupValidator(validatorType);

      if (validatorType.equals("multipart")) {
        if (validator instanceof MultiPartValidator) {
          MultiPartValidator multipartValidator = (MultiPartValidator) validator;
          readMultiPartElement(child, multipartValidator);
        } else {
          // if this occurs, there has been a major programming error
          throw new ValidatorException(
              "Validator is not a multipart validator!");
        }
      } else {
        readPartParameters(child, validator);
      }
      parentValidator.addValidator(childName, validator);
    }
  }

  /**
   * Read a part element and all of its supplied parameters.
   *
   * @param partElement
   *            the part element to read
   * @param validator
   *            the validator to load configuration parameters into
   */
  private static void readPartParameters(Element partElement,
      Validator validator) {
    NodeList paramList = partElement.getChildNodes();
    for (int j = 0; j < paramList.getLength(); j++) {
      Element param = (Element) paramList.item(j);
      validator.addParameter(param.getAttribute("name"), param
          .getTextContent());
    }
  }

  /**
   * Looks up a validator by type name.
```

```
   *
   * @param validatorType
   *            the type of validator
   * @return an instance of a validator
   */
  public static Validator lookupValidator(String validatorType) {
    loadListOfValidatorsLock();

    String validatorClassName = null;
    Lock readLock = lock.readLock();
    readLock.lock();
    try {
      validatorClassName = availableValidators.get(validatorType);
    } finally {
      readLock.unlock();
    }
    Validator validator = null;

    if (validatorClassName != null) {
      try {
        validator = (Validator) ClassLoaderUtil.loadClass(
            validatorClassName, ConfigParser.class).newInstance();
      } catch (InstantiationException e) {
        throw new ValidatorException(
            "Error while trying to load validator class of type: "
                + validatorType, e);
      } catch (IllegalAccessException e) {
        throw new ValidatorException(
            "Error while trying to load validator class of type: "
                + validatorType, e);
      } catch (ClassNotFoundException e) {
        throw new ValidatorException(
            "Error while trying to load validator class of type: "
                + validatorType, e);
      }
    } else {
      if (log.isWarnEnabled())
        log.warn("No validator loaded for the type '" + validatorType
            + "'");
    }

    return validator;
  }

  /**
   * Loads the list of validators. This method is thread safe.
   */
  private static void loadListOfValidatorsLock() {
    Lock readLock = lock.readLock();
    readLock.lock();
    try {
      if (availableValidators.isEmpty()) {
        // no validator types loaded, need to aquire writelock and load
        // list
        readLock.unlock();
        Lock writeLock = lock.writeLock();
        writeLock.lock();
        try {
          // need to recheck for null beacuse of possible interferring
          // threads
          if (availableValidators.isEmpty()) {
            // load the default list first
            loadListOfValidatorsNoLock(Constants.VALIDATORS_DEFAULT);
            // then load custom validators if it exists
```

```
            loadListOfValidatorsNoLock(Constants.VALIDATORS_CUSTOM);
          }
        } finally {
          readLock.lock();
          writeLock.unlock();
        }
      }
    } finally {
      readLock.unlock();
    }
  }

  /**
   * Does the actual loading of the validator list. This is not thread safe
   * and should only be called by
   * {@link loadListOfValidatorsLock loadListOfValidatorsLock}.
   */
  private static void loadListOfValidatorsNoLock(String configurationFile) {
    InputSource configSource = new InputSource(ClassLoaderUtil
        .getResourceAsStream(configurationFile, ConfigParser.class));

    if (configSource != null) {
      try {
        Document doc = DomHelper.parse(configSource);

        NodeList validatorList = doc.getElementsByTagName("validator");
        for (int i = 0; i < validatorList.getLength(); i++) {
          Element element = (Element) validatorList.item(i);

          String name = element.getAttribute("name");
          String implClass = element.getAttribute("class");

          availableValidators.put(name, implClass);
        }
      } catch (ValidatorException e) {
        if (configurationFile.equals(Constants.VALIDATORS_CUSTOM)) {
          // A failure during loading of custom validators is not
          // critical. Usually thrown because of a non-existing file
          if (log.isWarnEnabled())
            log.warn("Could not load validator configuration from "
                + configurationFile);
        } else {
          throw e;
        }
      }

    }
  }
}
```

Listing B.3: Source Code - no.fjas.wspv.ConfigParser

B.4 Source Code - *no.fjas.wspv.ValidatorSupport*

Listing B.4 shows the source code for the *no.fjas.wspv.ValidatorSupport* class.

```
package no.fjas.wspv;
```

```
import no.fjas.wspv.validators.MultiPartValidator;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

/**
 * Helper class for validating a SOAP envelope wrapped in a
 * {@link org.w3c.dom.Document}.
 *
 * @author <a href="mailto:henning _at_ fjas.no">Henning Jensen</a>
 */
public class ValidatorSupport {

  protected static Log log = LogFactory.getLog(ValidatorSupport.class);

  /**
   * Validates a SOAP envelope. Currently only supporting SOAP 1.1
   *
   * @param config
   *            the service configuration to use when validating
   * @param doc
   *            the entire soap envelope
   * @throws ValidationException
   *             upon validation errors
   */
  public static void validateEnvelope(ServiceConfig config, Document doc)
      throws ValidationException {
    String serviceName = config.getName();
    Element envelope = doc.getDocumentElement();
    if (envelope != null) {

      // Using SOAP 1.1 namespace
      NodeList bodyList = envelope.getElementsByTagNameNS(
          Constants.SOAP_NAMESPACE_11, "Body");

      if (bodyList != null) {
        Element body = (Element) bodyList.item(0);
        if (body != null) {
          if (log.isDebugEnabled())
            log.debug("body fetched: " + body.getNodeName());

          Element operation = (Element) body.getFirstChild();
          if (operation != null) {
            String operationName = operation.getNodeName();
            if (log.isDebugEnabled())
              log.debug("operation.name: " + operationName);

            NodeList partElements = operation.getChildNodes();
            for (int i = 0; i < partElements.getLength(); i++) {
              Element part = (Element) partElements.item(i);

              validateElement(config, serviceName, operationName,
                  part);
            }
          } else {
            if (log.isWarnEnabled())
              log.warn("No operation element found");
          }
        } else {
          if (log.isWarnEnabled())
            log.warn("No body element found");
```

```
        }
      } else {
        if (log.isWarnEnabled())
          log.warn("No body elements found within envelope");
      }
    }
  }

  /**
   * Validates a single element
   *
   * @param config
   *            the service configuration
   * @param serviceName
   *            the name of the service
   * @param operationName
   *            the name of the operation
   * @param part
   *            the part element to validate
   * @throws ValidationException
   *             upon validation errors
   */
  private static void validateElement(ServiceConfig config,
      String serviceName, String operationName, Element part)
      throws ValidationException {

    StringBuilder requestPath = new StringBuilder(serviceName);
    requestPath.append("/");
    requestPath.append(operationName);
    requestPath.append("/");
    requestPath.append(part.getNodeName());

    Validator validator = config.getValidators()
        .get(requestPath.toString());

    if (validator != null) {

      // finally, calling the validate method
      validator.validate(part);

      if (log.isDebugEnabled())
        log.debug("Validation successful");
    } else {
      if (log.isWarnEnabled())
        log.warn("No validator defined for " + requestPath.toString()
            + "(" + part.toString() + ")"
            + ", skipping validation.");

      if (config.isValidateEverything()) {
        if (log.isWarnEnabled())
          log.warn("Validation aborted due to missing validator for" +
              " element "  + requestPath.toString());
        throw new ValidationException("Invalid value");
      }
    }
  }
}
```

Listing B.4: Source Code - no.fjas.wspv.ValidatorSupport

Appendix C

Source Code - WSPV XFire Handler

Listing C.1 shows the source code for the WSPV XFire handler, which connects XFire to WSPV.

```
package no.fjas.wspv.xfire;

import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;

import no.fjas.wspv.ConfigParser;
import no.fjas.wspv.Constants;
import no.fjas.wspv.ServiceConfig;
import no.fjas.wspv.ValidatorSupport;
import no.fjas.wspv.util.ClassLoaderUtil;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.codehaus.xfire.MessageContext;
import org.codehaus.xfire.exchange.AbstractMessage;
import org.codehaus.xfire.handler.AbstractHandler;
import org.codehaus.xfire.handler.Phase;
import org.codehaus.xfire.util.dom.DOMInHandler;
import org.w3c.dom.Document;

/**
 * This handler connects XFire to WSPV. Handler is invoked during the PARSE phase.
 * Policy configuration is loaded from META-INF/wspv/validation.xml
 *
 * @author <a href="mailto:henning _at_ fjas.no">Henning Jensen</a>
 */
public class WSPVHandler extends AbstractHandler {

  protected static Log log = LogFactory.getLog(WSPVHandler.class
      .getName());

  private Map<String, ServiceConfig> globalServiceConfig = new HashMap<String,
      ServiceConfig>();

  public WSPVHandler() {
    super();
```

```
    loadConfiguration();
  }

  public WSPVHandler(String configurationFile) {
    super();
    loadConfiguration(configurationFile);
  }

  private void loadConfiguration() {
    loadConfiguration(Constants.DEFAULT_CONFIGURATION_FILE);
  }

  @SuppressWarnings("unchecked")
  private void loadConfiguration(String configurationFile) {
    // Make sure handler is in correct phase and after DOMInHandler
    setPhase(Phase.PARSE);
    getAfter().add(DOMInHandler.class.getName());

    InputStream is = ClassLoaderUtil.getResourceAsStream(configurationFile,
        WSPVHandler.class);
    if (is != null) {
      globalServiceConfig = ConfigParser.parseValidatorConfigs(is,
          configurationFile);
    } else {
      if (log.isWarnEnabled())
        log.warn("Validation configuration not loaded, config file not found.");
    }
  }

  public void invoke(MessageContext context) throws Exception {
    if (log.isDebugEnabled())
      log.debug("ValidatorInHandler has been invoked");

    String serviceName = context.getService().getName().getLocalPart();
    if (log.isDebugEnabled())
      log.debug("service.name: " + serviceName);

    ServiceConfig serviceConfig = globalServiceConfig.get(serviceName);
    if (serviceConfig != null) {
      if (log.isDebugEnabled())
        log.debug("Service configuration found for service: "
            + serviceName);

      AbstractMessage currentMessage = context.getInMessage();
      Document doc = (Document) currentMessage
          .getProperty(DOMInHandler.DOM_MESSAGE);

      if (doc == null) {
        log.warn("Document is null!");
      } else {
        if (log.isDebugEnabled())
          log.debug("Validating document");
        ValidatorSupport.validateEnvelope(serviceConfig, doc);
      }

    } else {
      log.warn("No validation configuration exists for service: "
          + serviceName);
    }
  }
}
```

Listing C.1: Source Code - no.fjas.wspv.xfire.XFireValidatorHandler

Appendix D

Source Code - WSPV Axis2 Handler

Listing D.1 shows the source code for the Apache Axis2 validation handler, which connects Axis2 to WSPV.

```
package no.fjas.wspv.axis2;

import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;

import no.fjas.wspv.ConfigParser;
import no.fjas.wspv.Constants;
import no.fjas.wspv.ServiceConfig;
import no.fjas.wspv.util.ClassLoaderUtil;
import no.fjas.wspv.ValidatorSupport;

import org.apache.axiom.soap.SOAPEnvelope;
import org.apache.axis2.AxisFault;
import org.apache.axis2.context.MessageContext;
import org.apache.axis2.engine.Handler;
import org.apache.axis2.handlers.AbstractHandler;
import org.apache.axis2.saaj.util.SAAJUtil;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.w3c.dom.Document;

/**
 * This handler connects Axis2 to WSPV.
 * Policy configuration is loaded from META-INF/wspv/validation.xml
 *
 * @author <a href="mailto:henning _at_ fjas.no">Henning Jensen</a>
 */
public class WSPVHandler extends AbstractHandler {

  protected static Log log = LogFactory.getLog(WSPVHandler.class.getName());

  private Map<String, ServiceConfig> globalServiceConfig = new HashMap<String,
      ServiceConfig>();

  public WSPVHandler() {
    super();
```

```
    loadConfiguration();
  }

  public WSPVHandler(String configurationFile) {
    super();
    loadConfiguration(configurationFile);
  }

  private void loadConfiguration() {
    loadConfiguration(Constants.DEFAULT_CONFIGURATION_FILE);
  }

  private void loadConfiguration(String configurationFile) {
    InputStream is = ClassLoaderUtil.getResourceAsStream(configurationFile,
        WSPVHandler.class);
    if (is != null) {
      globalServiceConfig = ConfigParser.parseValidatorConfigs(is,
          configurationFile);
    } else {
      if (log.isWarnEnabled())
        log.warn("Validation configuration not loaded, config file not found.");
    }
  }

  public InvocationResponse invoke(MessageContext context) throws AxisFault {
    SOAPEnvelope envelope = context.getEnvelope();
    try {
      Document doc = SAAJUtil.getDocumentFromSOAPEnvelope(envelope);
      String serviceName = context.getAxisService().getName();

      ServiceConfig serviceConfig = globalServiceConfig.get(serviceName);
      if (serviceConfig != null) {
        if (log.isDebugEnabled())
          log.debug("Service configuration found for service: " + serviceName);

        ValidatorSupport.validateEnvelope(serviceConfig, doc);
      } else {
        if (log.isWarnEnabled())
          log.warn("No validation configuration exists for service: "  + serviceName)
              ;
      }
    } catch (Exception e) {
      e.printStackTrace();
      throw new AxisFault(e);
    }

    return Handler.InvocationResponse.CONTINUE;
  }

}
```

Listing D.1: Source Code - no.fjas.wspv.Axis2.WSPVHandler