XMLbroker: An XML-Based Web Integration Middleware Prototype for the J2EE Platform using SOAP

Design Outline

Jean-Marc Rosengard

Nov 2001

Contents

List of Figures

List of Tables

Code Listings

Chapter 1

Introduction

1.1 Problem Origin and Statement

1.1.1 The State of Inter-Enterprise Integration

In a response to the ever existing need to integrate new applications with existing ones *inside* the enterprise, the industry has launched dozens of different types of middleware—that is, software adapters which role is to abstract differences between information systems. Whenever there was a need for information systems from *different* companies to collaborate to achieve their business goal (an obvious example is the banking industry), some industries have organised vertically to create proprietary information interchanges and networks (e.g. Electronic Data Interchange, or EDI). Unsurprisingly, these technologies are characterised by a prohibitive access cost and are only approachable to large corporations, within well-structured markets.

In the recent years, the Internet has become widely available—both as a worldwide federated network and as a set of standard protocols. The Web has however long waited to be fitted with a data representation mechanism suitable for automated transactions. HTML, by essence presentation oriented—and made inextensible by design—proved inadequate for the development of automated web transactions. XML, on the other hand—as a meta-language, was inherently more suitable to take over this role. XML was originally destined to document publishing. Its attached data typing mechanism (DTDs), although probably sufficient for the original purpose of the language—proved inadequate for its new role as a middleware. The recent adoption of XML Schemas has lifted this deficiency.

As we will discuss in Section 1.3, several novel XML-based technologies effectively allow the cost of integrating applications across corporate borders to be potentially decreased by several *orders of magnitudes*—and consequently make inter-enterprise integration much wider spread than it is today. If such a dramatic cost fall occurs, there is no reason why applications should be *only* integrated between existing business partners. The applications could as well be exposed to the web *proactively*, as a service for potential new partners to use, rendering effectively software a *commodity*.

1.2 System Presentation

XMLbroker is the prototype of a middleware framework—based on the RPC paradigm—aimed at the integration of software over the Internet. We re-

fer to this concept as *Web integration* because further than embracing the Internet as a transport media, XMLbroker adheres to technology standards defined by the World Wide Web Consortium (W3C). Namely HTTP 1.1, and SOAP 1.1.
XMLbroker builds upon the server-side Java software infrastructure (Java 2 Enterprise Edition, or J2EE), exposing Enterprise Java Beans (EJB) components as Web services. The software is composed of two entities:

1. a server built as an extension module to the JBoss application server (e.g. running in the same process)

2. a client, implementing the transparency layers between the host application and the server

The reasons that led to the choice of the Java platform will be discussed in Chapter 3.

1.3 Core Technologies

1.3.1 The XML Meta-Language

The eXtensible Markup Language (XML) is a W3C standard [**?**] for defining markup-based data representation languages. With the recent addition of the XML Schema facility [**?**], XML-derived languages can be both processed and reified as XML.
XML—and consequently XML-based languages, are *well-formed* with respect to the XML structure definition. The notion of well-formedness refers to its acception in *predicate logic*. Well-formedness obviously facilitates (or even enables) an automated and uniform parsing of the language. Character representation in XML uses by default the Unicode standard. This, as well the elements above make of XML the *lingua franca* of data representation.

1.3.2 WSDL

As developed in Section **??**, the Web Service Description Language (WSDL) is a language—defined in XML—that is conceptually close to the Interface Definition Language (IDL) found in CORBA. In the Web service paradigm discussed in the last section, the role of WSDL is to expose a set of operations of a software component as a service. In order to be utilised by potential users, each WSDL document must be published, ideally in a UDDI registry (c.f. Section **??**) or as a simple URI.

The role of WSDL is paramount to the inter-operability of different of Web service frameworks. However, and in spite of its important role, it would be well outside the time-scope of this project to implement even part of the WSDL specifications. An augmented version of the Java interface will be utilised instead, to define a remote service. This role is referred to as *service interface* in the remainder of this report.

1.3.3 SOAP

The Simple Object Access Protocol (see Section **??**) represents one of the fundamental technologies constitutive of this project. The SOAP 1.1 recommendations cover three different topics.

1. a message encapsulation mechanism
2. a data representation scheme
3. a Remote Procedure Call (RPC) mapping to the HTTP protocol

The prototype built during this project (code-named XMLbroker) addresses all three areas. The limited scale of this project however means that the standard will only be implemented up to the degree of functionality necessary to provide a *qualitative appreciation* of the technology. A demonstration application—exposing a Network Time Protocol (NTP) server as a Web service, will support our discussion in Section A. We review below the main elements of the SOAP recommendations.

The SOAP message exchange model does not define an object brokering architecture, but a simpler RPC model that specifically does not support distributed garbage collection. By way of consequence, it does neither define passing object references nor object activation (which requires the ability to pass objects by reference).
These design decisions were taken on the grounds of both simplicity and flexibility. Needless to say that this project would not have been feasable otherwise.

The purpose of this chapter was to introduce the reader to the realm ot this project. We first presented the inception of the Web service paradigm, which will be further developed in the first part of Chapter **??**. We then outlined some more pragmatic aspects of this project, related to the engineering of the framework prototype. The last section was given as a compendium to the respective formal documents from the W3C—capturing only the items

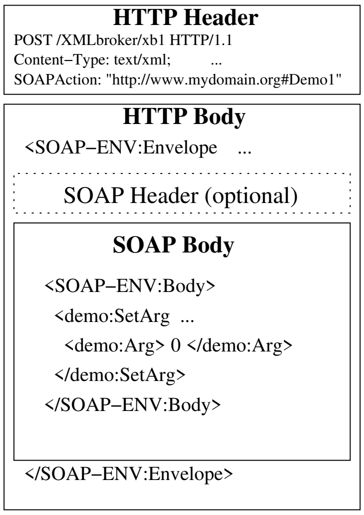

Figure 1.1: Structure of a SOAP message

SOAP Envelope The envelope defines the part of the host message that encloses the SOAP data. The envelope contains an optional header, and a mandatory body. The RPC-related data is part of the SOAP body.

SOAP Encoding The SOAP encoding system is a superset of the one defined by the XML Schema 1.0 specifications. Both simple types (string, number,...) and composite types (array, structure,...) may be used.

Mapping to HTTP The mapping to HTTP covers both the RPC mechanism, and the extensions to this protocol. A method call is specified by an XML element—parameters being child elements. A method call response element bears the same name as the call element, with the added *Response* suffix. Method calls and call responses are mapped respectively to HTTP POST requests and HTTP POST responses. A last important consideration is the adjunction of the *SOAPAction* field to the HTTP request header—which identifies the target service on the remote host.

of direct relevance to this project. The reader is referred to Chapter **??** and the documents themselves [**?**, Box, Ehenebuske, Kakivaya, Layman, Mendelsohn, Nielsen, Thatte & Winer 2000, **?**, **?**, **?**] for a detailed coverage of the subjects.

Chapter 2

Requirements Analysis

In this chapter, we present a refined model for XMLbroker, and then attempt a first approach to defining conceptual classes. To assist us with formulating the definition of the system in an explicit and unambiguous fashion, we adopt the formalisms of the Unified Modelling Language (UML).
Requirement specification documents are generally not presented with enough precision and coherence to be directly usable. In the present case however, the problem space is well defined by the W3C documents. By 'well defined' we mean presented in a precise, coherent and unambiguous fashion. Whenever necessary, W3C documents make use of a *formal notation* (the augmented Backus-Naur Form) to achieve this. We are therefore left with the tasks of defining those parts of the W3C specifications that can realistically be implemented in a project of this time-scope—and of formulating any addition we make.

2.1 Software Specifications

Most software applications present a major *operational* facet and a minor *configurational* facet. Whereas the former is addressed by the standards, the latter is implementation specific.

2.1.1 Software Configuration

No run-time configuration is supported. The deployment of services is performed at startup, from a 'service.conf' file. The configuration of XMLbroker server module is accomplished at the same time by the startup shell script. In the client module, the standard Java mechanism of '-D' command line options is used to define the key-value pairs associated with a remote service. As formulated in Section 2.1.3, SOAP fault messages are not supported. We must therefore provide an alternative means to identify remotely whether XMLbroker server is running. An HTML page displaying 'XMLbroker is ready for service' is sufficient.

2.1.2 The Hypertext Transfer Protocol

SOAP 1.1 does only require the HTTP POST method. So only POST is supported by the client module. The server module must however also implement the GET method, to handle a GET request from a standard browser—and respond with a server-is-up message. No HTTP fault code support is required.

2.1.3 The Simple Object Access Protocol, 1.1

SOAP Envelope XMLbroker does not support any optional feature of SOAP. This applies in particular to the SOAP header.

SOAP Encoding Only the encoding of character strings is implemented.

SOAP RPC Only point-to-point messages are handled, not chained messages (e.g. SOAP Actors are not implemented). SOAP fault messages are not supported.

2.2 Requirement Formalisation

A general order consideration is standing out of the diagram in Figure 2.1: the number of use cases (five) is small, but the complexity of each operational use case is high. This is an essential feature of frameworks, which must achieve a substantial amount of processing to handle a complex aspect of the system on behalf of the client application. In the case of middleware frameworks in general—and of XMLbroker in particular, this complexity is defined by the need to provide a suitable level of *transparency* for remote method calls.

We will now review each use case in some detail, starting with their three

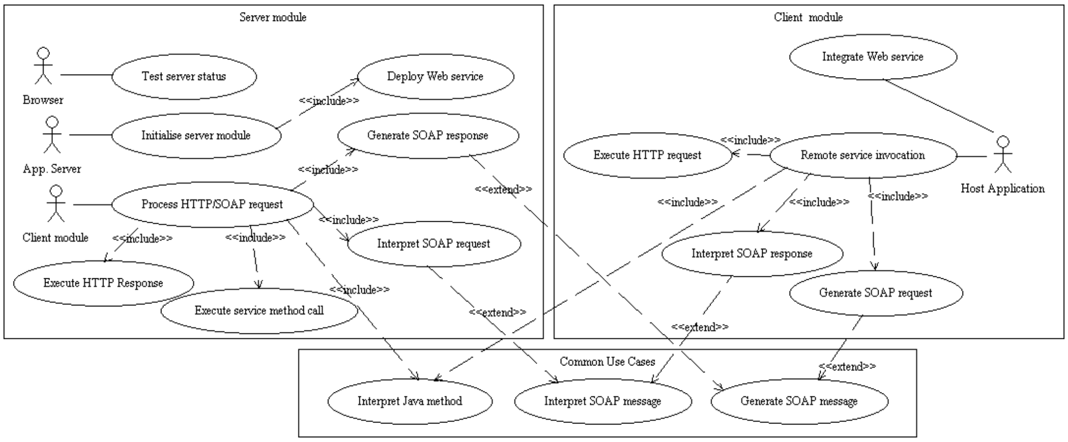

Figure 2.1: Global view of use cases

fundamental constituents: SOAP interpretation, SOAP message generation and Java Interpretation.

2.2.1 Common Use Cases

SOAP Interpretation The first logical unit of labour given a SOAP message is XML parsing. This process is delegated to the Apache Xerces XML parser, which turns a data stream into a data structure. The SOAP semantics are then applied to the data structure—leading if the process is successful to:

1. a method name and a list of actual input parameters, in the case of a SOAP request
2. a method name and a list of actual output parameters, in the case of a SOAP response

If the interpretation process fails, the reason for the failure is logged.

SOAP Message Generation Message generation is the reverse process of interpretation. Given the elements listed above, and the service's meta-data, a SOAP message is devised using the SOAP envelope and encoding rules.

Java Interpretation As we have already outlined in Section 1.3.2, we will utilise a superset of the Java interface to define services (i.e. remote methods). This is rendered necessary by the need to resolve the semantic mismatch between Java and SOAP method signatures: the absence of a formalism to indicate output parameters in Java. Whereas the discussion of the solution to this problem is the subject of Section 4.2, we are left with the need to extract the constitutive elements of a Java method signature—that is, a *method name* and a set of *formal parameters*, together with their *passing mode*. Notice the similarity with the output of the SOAP interpretation process.

2.2.2 Server Subsystem Use Cases

Initialisation of the Server Subsystem The server module initialisation is launched by the application server—whenever the latter is explicitly started. The service deployment takes place at this time, and three elements must be defined for each service: the *service name*, the *service interface* and a *servant*.
The service name is a Uniform Resource Identifier (URI), as define in the SOAP specifications [Box et al. 2000].
The service interface defines the method name, the parameter type and the parameter passing modes—the parameter names are not important but the

order of parameters is (see ibid.). Each service interface must extend the *XbService* marker interface. The reader is directed to [Arnold & Gosling 1998] for a definition of marker interfaces.
The servant is a *delegate* JavaBean that is reponsible for invoking the EJBs that actually implement the service. In very simple cases, the delegate may implement the service itself.

Service Method Invocation As we have seen, the abstract representation of method signatures is a key element in XMLbroker. Each service when deployed will be attached a *formal signature*—devised from the Java interpretation process described above. Whenever a SOAP request is received, the interpretation of the message leads to an *actual signature*. A Java method call is then prepared and executed (c.f. Figure 2.2) by reconciling the formal and actual method signatures.

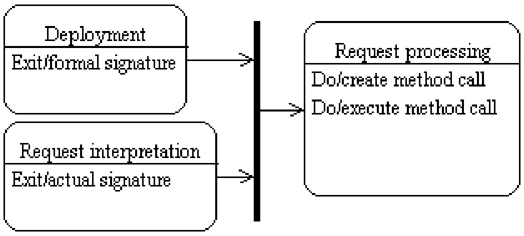

Figure 2.2: State transitions in the method call process

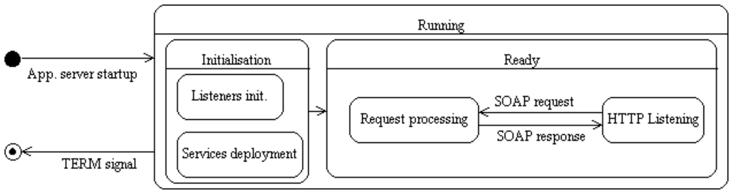

Figure 2.3: Dynamic model of the server subsystem

2.2.3 Client Subsystem Use Cases

Service Integration Services may be integrated in a client application in two simple steps. The remote service is first defined with a host name, a service name and a service interface. The service interface is then—once again, utilised to devise an abstract representation of the remote method's signature.

Remote Service Call The task of using a remote service is made as straightforward as executing a local method call of the form:

`outParameters=XbProxy.invokeService(inParameters)`. A collection of input parameters is passed to an instance of the client module (traditionally called a proxy or stub in this context)—and a collection of output parameters is returned.

2.3 Mapping to an Object Model

An important aspect of the use case model presented above is the predominance of a few use cases, common to both the client and the server subsystems. This commonality is expressed with both the *inclusion* and *extension* semantics. As a consequence, the class models (Figure B.1) for both modules share most of their elements.

Two classes are however noticeably different: the HTTP networking class, and the SOAP message representation and generation class. As discussed in Section 2.1.2, the client module only requires a lightweight implementation of HTTP POST. On the other hand, the server must provide a full implementation of the protocol—and support concurrent connections.

The class `Message` encapsulates the representation and creation process of the generic elements of a SOAP message. Two subclasses—`ReqMsg` and `RespMsg`—complement it with the elements specific to request and response messages.

Chapter 3

Software Architecture

Whereas in the past chapter we have concentrated on defining a system model in the problem-space, we now introduce the global structure of XML-broker in the context of the proposed software platform and discuss how the diverse components of the system interact.

3.1 Software Infrastructure

3.1.1 Operating Environment

As we will discuss in Section 3.2, XMLbroker is not an autonomous piece of software. It is part of a larger middleware platform. Integrating modules from different origins into a cohesive software entity is not a straightforward exercise and it requires some understanding of the underlying operating system environment and services. The GNU/Linux operating system was adopted because it provides an open, stable and well understood platform. This decision is part of a broader strategy to diminish the risk associated with the use of technology—in order to allow a linear-time development (see Section **??**).

3.1.2 Programming and Deployment Platform

We will now review the core technologies used both inside, and in conjunction with XMLbroker—and justify why a technology has been favoured against another.
Table 3.1 presents some candidate programming languages with their associated environments.

Language	**HTTP networking**	**Business objects**	**Support for middleware**	**Language experience**	**Productivity**
C++	Libwww, CGI	through CORBA	templates, native to OS	⋆ ⋆ ⋆	⋆
Perl	CGI module	–	reg. ex., shell integr.	⋆	⋆ ⋆ ⋆
Python	diverse modules	Zope products	reflection, Zope app. serv.	⋆	⋆ ⋆ ⋆
Java	servlet API	EJB	J2EE, reflection, classloader,...	⋆ ⋆ ⋆	⋆ ⋆

Table 3.1: Semi-quantitative comparison of programming languages and their APIs. XML parsing is not considered a discriminating factor because of the wide availability of parsers for all the languages considered.

The first two columns, labelled *HTTP networking* and *business objects*, correspond to the two levels of 'plugability' of XMLbroker into its environment. Whereas these will be discussed in the next section, we only consider here their level of support by the APIs associated with the different languages. The column labelled *support for middleware* displays the features associated with the languages, that present a particular interest in the context of middleware—mainly by allowing meta-programming (c.f. Section **??**). The last column presents a productivity rating associated with the diverse languages. It is adapted from [Prechelt 2000] and is given irrespectively of any experience with the languages.

C++ Mainly because its maturity, the C++ language features many third party libraries. Many of them are however proprietary, like Roguewave class libraries. Open source networking libraries however abound—chief among them being the W3C Libwww, the HTTP reference implementation. However, the only non-proprietary architecture for business-level components is the CORBA component architecture, provided by open source CORBA implementations like MICO and TAO.
C++ support for *generic programming* (e.g. templates) means that a function (or class) may be programmed using *parameterised types*. The compiler will then be in charge of substituting the generic type with the actual type passed to the function. Mainly because of my experience with the language, C++/CORBA was the initial, a prori solution.

Perl Perl is by essence the language of choice for character data manipulation—and in the present case XML data manipulation. HTTP network programming is also widely supported in Perl. However, the language fails to offer any element of component architecture.

Python Like Perl, Python is a 'scripting language'. It is a fully featured object-oriented language, which even offers some reflective capabilities (see Section **??**). Most importantly, Python features the Zope application server, and its extension modules (called *products*). Although Zope is primarily aimed at content management, it is extensible and made of Python a credible alternative to Java.

Java Java was chosen for several reasons. First it offers a simple, object-oriented and reflective language. But its main strength lies in the J2EE platform (see Figure 3.1). Among the various APIs offered by J2EE are the *Servlet* API, and the *Enterprise Java Bean* (EJB) component framework.

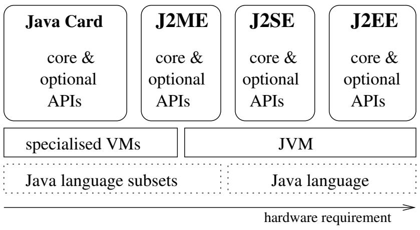

Figure 3.1: The Java 2 platform

The Java 2 Platform exists in four versions, each targeted at a different hardware environment. The Java Card, is the platform for smart card applications. The Micro Edition (J2ME) is aimed at the embedded devices market (phones, cars,...). The Standard Edition (J2SE) is provided for building desktop applications. Finally, the Enterprise Edition (J2EE) is aimed at the server-side enterprise applications market.

These form the foundations on which XMLbroker server is plugged into its software environment. J2EE represents however only a set of specifications, fixed by the Java Community Process (JCP). These are then implemented in J2EE application servers.
This project initially started using IBM WebSphere 3.5 application server. Websphere however presented some deployment issues, due to its partial compliance to the J2EE specifications. The JBoss application server—an open-source implementation of J2EE, was adopted instead.

We concluded Chapter **??** by the proposition that components collaborating through XML adapters could be the building blocks of a new generation of distributed applications. This view is matured in this chapter, by introducing the *Enterprise Java Bean* component architecture. In our model for exposing business services to the Web (see Section **??**), each service is implemented as one or several components (a.k.a. business objects). As formulated in the last section, the availability of the EJB component architecture is the primary reason for our choice of the J2EE platform. Whereas the detailed study of the EJB component model is not the subject of this report, some elements are worth noting:

- each EJB component is executed within a module of the application server known as an *EJB container*
- components only provide the *business behaviour* of a service
- they delegate the *system behaviour*, that is; the implementation of the transparency layers, resource management and transaction management to the EJB container and other modules of the application server.

3.2 Application architecture

3.2.1 Logical Structure

Figures 3.2 and 3.3 represent the high-level structure of XMLbroker and its environment, and the logical relationships between their elements. The model can be partitioned into two conceptual layers; the business layer and the system layer.

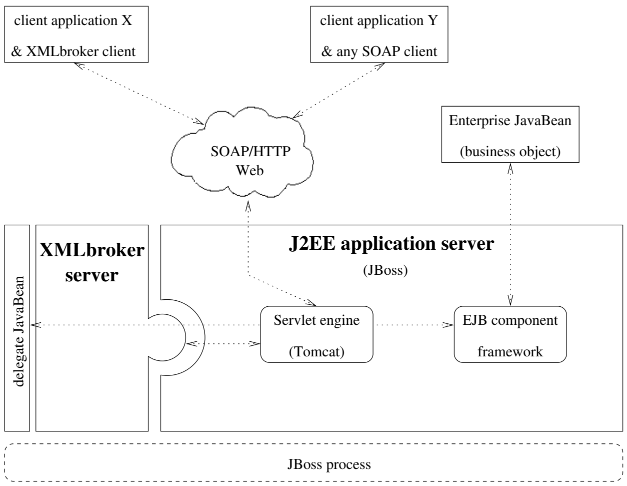

Figure 3.2: High-level architecture of XMLbroker

Business Layer The business modules have been presented having two roles: client application, and server component. It is however important to note that this client-server dichotomy represents only a particular case. The general case is that both roles are symmetric and dynamically interchangeable—involving two instances of the model shown. This simplification is however maintained in the context of the demonstration application, presented in Appendix A.

System Layer The system layer contains the actual middleware. These are XMLbroker server module, the application server with its EJB container and the Servlet engine. On the client application side, XMLbroker client module has the role of a *proxy*, abstracting the remote service to a local Java object. The *delegate* Java Bean holds a 'glue' role between the a particular web service deployed on XMLbroker and set of Enterprise Java Beans, while preserving a level of indirection between them. As such, it represents the interface between the two layers.

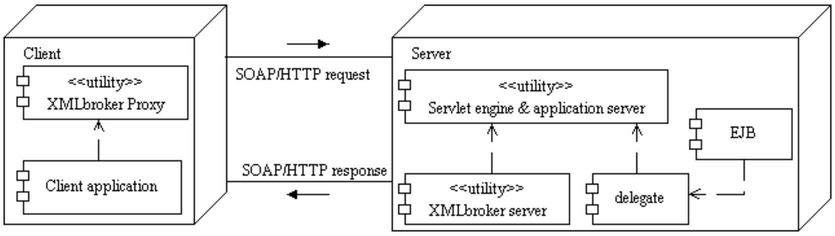

Figure 3.3: UML deployment diagram for the proposed system

3.2.2 Components Taxonomy and Collaboration

Client Proxy The client proxy (or stub) is a lightweight, independent, portable and single-threaded software entity. Unlike most RPC systems, XMLbroker stubs are *dynamic proxies.* They are not generated for a particular service interface, but *adapt* to any valid service interface at run-time. Whereas this approach allows a more simple and flexible Web service integration, there exists a slight performance tradeoff compared with the traditional static approach.

Server Module The server module must be 'plugged' into the J2EE platform—delegating HTTP handling to the Servlet engine (Apache Tomcat) that provides a multi-threaded HTTP listening facility of 20 concurrent listeners. For this purpose Tomcat has been configured to support SOAP, and the base class `HttpConnector` extends the Servlet API.
An outcome of the concurrency in the server module is the need to adopt a safe resource sharing strategy. A quick review of the arity of the class associations in Figure B.1 reveals that only the objects encapsulating the services' meta-information (e.g. instances of `Port`) need to be shared. This is the result of an architectural decision to apply a stateless object collaboration model sometimes documented as the *stateless pattern.* In this model most data elements are passed to the object on method call and objects do rarely share any data, removing the need for data integrity protection mechanisms, therefore leading to improved performance. This approach is utilised in many application servers, including JBoss. The model may appear to go against object-orientation, it is however essentially applied to transactional objects—that is, objects modelling a transaction as opposed to objects modelling an

domain entity.
The module is not portable for two reasons. First the JBoss startup script requires the Bourne shell. Moreover, the `NativeHelper` class, a wrapper for the `libXMLbroker.so` shared-library, may be used to generate uniques servant identifiers.

Chapter 4

Elements of Design

We discuss in this chapter the low level aspects of the software solution to the domain model presented in Chapter 2. The reader is referred to the figures B.2 and B.3 for a global view of the refined class model.

4.1 Reflection as the Basis of Dynamic Proxies

We introduced in Section ?? the concept of *computational reflection* and its partial support in the Java language. In the next two sections, we present two elements of XMLbroker that rely on the Java reflection facilities: the implementation of stubs as *dynamic proxies* and the semantic extension of the Java method signature.

Traditionally, the data encoding/decoding (also called marshalling/unmarshalling) logic specific to a particular distributed object's interface is built into the client and server stubs—each materialised by a class file, generated by a dedicated compiler. In Java, this may be achieved by first creating a source code tree, serialising it into a character stream, and then invoking the java compiler via the `sun.tools.javac` package.
Java however offers a more elegant solution by the means of reflection. At the root of the inheritance tree is the `java.lang.Object`, containing the `getClass()` method—returning an instance of the `java.lang.Class` meta-class. Given any object or interface name, it is therefore possible to obtain an instance of the meta-class, and *discover* all the visible properties of the object. In the example given in Listing ??, the return type of a service method is checked. In our model, output parameters are supported, rendering returned values non-necessary and therefore not supported. This example is adapted from the `validate()` method of the `JavaInterpreter` class.

```
// get a meta-class from the interface name.
Class cls=Class.forName(serviceInterfaceName);
// get all meta-methods from the meta-class
Method methods[]=cls.getMethods();
// get the return type for the first method
if(methods[0].getReturnType()!=void.class) throw new JavaIprException(
 cls.getName()+" : only the 'void' return type is allowed");
```

Code Listing 4.1: Example validation of a service interface using reflection

It is also possible to call a method from its meta-method object. This feature is utilised in the `Linker` class to build and then invoke the service method on the servant object. Listing ?? presents a simple example of such a method invocation mechanism.

```
A a=new A();
Method meth=a.getClass().getMethods()[0];
String param=new String("Hello");
meth.invoke(a,param);
```

Code Listing 4.2: Example method invocation using a meta-method object

4.2 The Interpretation of Service Interfaces

As we have discussed in Chapter 1, the realism of the limited scope of this project has constrained us to use Java interfaces in place of WSDL. Java method prototypes however need to be complemented to bridge the semantic gap beetween WSDL and Java. What we mean to imply here, is not the semantical deficiency of the Java interfaces—but rather that the *interpretation* of those semantics is disparate. As obvious as this statement may seem, Java interfaces presuppose the Java object model. WSDL on the other hand do not assume object-orientation at all—leaving out a particular object meta-model.

4.2.1 Parameter Passing Modes

A first practical element of the issue just discussed is the interpretation of function parameters. Whereas most high-level languages provide at least one mechanism to pass arguments by reference, very few of them address this issue in an expicit and prescriptive fashion.
In C, passing a structure `S` by reference is achieved by handing in a pointer to `S`. Now, the same syntax could be adopted for different reasons. One could be to avoid the overhead of passing a large structure by value, with no intent to alter it. Another could be the necessity to return multiple parameters. Yet another reason would be motivated by the need to actually alter the source structure. These semantically different usage of the language's syntax are not reflected by any syntactic formalism in the language.
C++ goes one step in this direction with the "pointer to const" and "const reference" constructs. In Java, non-primitive types (i.e. objects) are *implicitly* passed by reference. The interpreter itself can resolve these issues because it is able to detect if an element of data may be changed or not in the course of the programme's execution. This is however not the case of third-party programmes, which need an explicit indication of the intent of the programmer. In Ada—where nothing is implicit, programmers must signal their intent by using the `in`, `out` or `in out` argument passing mode keywords. The same applies to CORBA IDL. In the latter case as well as

in the case of WSDL and SOAP the need is dictated by the one-to-many relationship between the interface and implementation function prototype models.

4.2.2 Method Prototype Extension Scheme

Having demonstrated the need for the extension of the Java method signature, in order to support both input and output parameters, we are left with the task of devising a suitable mechanism. In a non-reflective language, the solution would be likely to involve both a set of keywords (e.g. `in` and `out`) and a *pre-processing* phase during which these would be interpreted and removed—before the java compiler be invoked.
Java however offers a more elegant alternative using, once again, the reflective capabilities of the language.

Let us define the following scheme:

1. the valid parameter passing modes are `__in` and `__out`
2. for each function parameter, its mode must be appended to the function name
3. modes must be specified in the same order as their corresponding parameter
4. all input parameters must precede the first output parameter in the definition sequence

An example of such a method prototype is given in Table 4.1. All method

```
void getBookDetails__in__out__out(String isbn,String author,String title);
```

$$\Downarrow$$

Parameter name	Passing mode	Parameter type
isbn	input	String
author	output	String
title	output	String

Table 4.1: Example of an extended method prototype

prototypes resulting from this scheme will be valid with respect to the Java language. No pre-processing stage is therefore required. The method name can be obtained by reflection, and parsed in order to reconcile the parameters with their respective passing mode. This is the role of the `ModeQueueBuilder` class (c.f. Figure B.2).

4.3 HTTP Client Implementation

As discussed in Section 2.1.2, only a partial implementation of the HTTP protocol is needed in the client proxy. The core Java class `java.net.HttpURLConnection` provides an implementation of HTTP. The corresponding API is however meant to be used at the URL level—abstracting the HTTP request/response mechanism.
The HTTP header of a SOAP request message (see Section 1.3.3) must contain the `SOAPAction` field. The Java API does however not support the addition of new (e.g. non-HTTP) fields to the HTTP header.
For these two reasons, those parts of the protocol required by the XML-broker client module have been implemented—from TCP sockets—in the `HttpClient` class.

4.4 XML Parsing Stategy

Two XML parsing strategies are commonly utilised: the W3C Document Object Model (DOM) and the Simple API for XML (SAX).

4.4.1 The Simple API for XML

SAX a is a sequential, event driven, stateless (and therefore read-only) parsing strategy. The client application registers its interest in selected XML element and defines the respective call-back methods. When parsing occurs, the document is processed sequentially and the relevant call-back method is invoked whenever the corresponding XML element is encountered. During this process, no record is made of the elements already parsed. Whereas this may lead to *performance benefits*, the whole operation has to be started again if an element needs to be accessed at a later stage.

4.4.2 The Document Object Model

Unlike SAX, DOM is a statefull parsing strategy, and it consequently also represents a data structure. During parsing, new elements are added to a tree data structure, leading to a memory representation (DOM tree) of the integrality of the XML document. The DOM tree may then be traversed in any suitable fashion, and nodes may be altered if necessary. This flexibility may however have an important performance overhead, both in term of memory and processing requirement—depending on the size of the document.

SOAP messages are typically small documents, of in the order of 10 to 50 nodes, so that performance degradation is not an issue. Two more arguments are in favour of the use of the DOM strategy. First, all of the elements of the DOM tree are needed during the SOAP interpretation process. Then there is also a need to traverse the tree back-and-forth. Because of the flexibility offered by the DOM approach, it is also used during SOAP messages construction, by the `Message` class and its subclasses.

4.5 Design Patterns

We recognised in Section **??** design patterns as an essential means to reuse successful designs. Most patterns introduce in the design a level of flexibility that is of no use to a project of this scope. Only two of the 23 fundamental patterns described in [Gamma, Helm, Johnson & Vlissides 1995] have consequently been used; the *singleton* pattern, and the *template method* pattern. The latter has been implemented (i.e. adapted from C++, since this language is used in [Gamma et al. 1995]) at two occurrences.
The *command* pattern could have been introduced in the design of the SOAP interpretation process, should the support for encoding more data types have had to be provided.

4.5.1 The Singleton

The meta-data relevant to each deployed service is held in `Port` objects. These objects must be shared between the 20 *concurrent workers* (i.e. threaded request processors). This role has been assigned to a `PortRegistry` object.

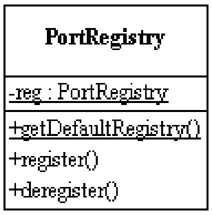

Figure 4.1: The Singleton pattern

For the desired effect to occur, a *single* instance of `PortRegistry` must be shared: this is the role of the Singleton pattern.
To prevent the creation of more than one instance, all constructors are `private`, and the class is `final`, so it may not be subclassed. The class method (i.e. `static` method in Java) `getDefaultRegistry()` returns a reference to the private class member `reg` of type `PortRegistry`—instantiating `reg` the first time it is invoked.

4.5.2 The Template Method

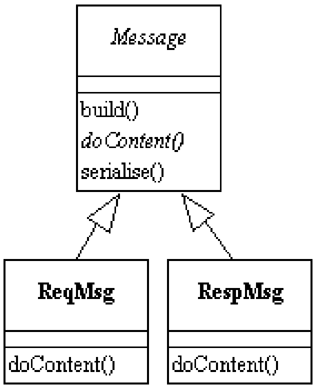

Figure 4.2: The Template Method pattern

The `Message` class encapsulates the data (e.g. DOM tree) and the necessary operations to build a generic SOAP message. Its subclasses `ReqMsg` and `RespMsg` *extend* both its knowledge and behaviour to reflect the *singular aspects* of both request and response message types.

Because the `doContent()` abstract method of the base class does not as such provide any default implementation, both subclasses must implement it, providing the behaviour necessary to build the actual content of the message. This behaviour is specific to a message type.

The `build` method of the base class is invoked to create the SOAP header and part of the SOAP body. It then calls the relevant implementation of `doContent()` to complete the message.

Chapter 5

Implementation and other Engineering Considerations

5.1 Build Process and Runtime Initialisation

5.1.1 Build Process

Although `javac` can resolve class files dependencies, this does not (and cannot) apply when classes are only discovered at runtime[1]. Also, no provision is made for other types of files, mainly C source files and Javadoc documentation files (see Section 5.4). The standard `make` utility, together with a hierarchy of makefiles, was therefore preferred. A shell script (build.ksh) allows according to the options provided, to either build the software for testing or to release. In the latter case, the Javadoc documentation is also generated. An option also allows to run the unit tests in batch mode.

5.1.2 Runtime Initialisation

Another script (run.ksh) is provided to setup the environment (i.e. classpath,...) necessary to run individual unit and integration tests.

5.2 Performance Considerations

Software developed with the Java platform often suffers from performance problems. When discussing this issue, it is important to recognise two facets in the problem: the performance overhead inherent to the facilities offered by the platform, and the fruitless overhead associated with a poor utilisation of those facilities.

5.2.1 Intrinsic Performance Issues

Java is a simple, object-oriented, portable, strongly-typed and boundary-safe language—with a built-in automated garbage collection mechanism. These features clearly result in a higher programmer productivity—of around 40–50% compared to C++ according to [Prechelt 2000]. On the other hand, [IBM Research 2000] suggests that the associated performance loss is in the order of 20%.
Whereas the latter figure is probably acceptable for most business application code (e.g. EJBs, servlets), it may not be the same for system-level code (e.g. JVMs, application servers). In this case, the *critical section* of the code may be optimised by pooling/recycling objects, using custom data-structures,... up to rewriting some of the code in C and integrating it via the

[1] this concept is referred to as *dynamic adaptability* in [Sametinger 1997]

Java Native Interface (JNI).

A consideration which importance is often underestimated is the choice of a Java Virtual Machine (JVM). Although this behaviour has obviously changed in the past few years, the JVMs from Sun Microsystems where mainly targeted at client applications. An example of this was the implementation of an *asynchronous* garbage-collector [Hunt & McManus 1998]; allowing the user threads to be *suspended* while garbage-collection was taking place. With the migration of Java to a server-side programming language, the situation has much improved. In particular, the old adage "Java is slow because interpreted" is now almost irrelevant, with the adoption of technologies such as Just In Time (JIT) compilation and Sun's HotSpot Server. The Java vendor TowerJ now sells a high-performance, server-side-only JVM.

We use in this project an IBM virtual machine (IBM Java 2, 1.3 for Linux) since it is the best performing free JVM for Linux, according to the Volano benchmark (the reference benchmark for JVMs).

5.2.2 Extrinsic Performance Issues

A great strength of the language is its simplicity. Or more truthfully, it is very easy to write inefficient code in Java. A more thorough understanding of the language is however necessary to devise reasonably efficient code.
For instance, the `Vector` class, which is probably the most widely used collection class, is thread-safe (i.e. declared `synchronized`). The protection mechanism has a performance cost that may of course be justified in a multi-threaded environment, but not in a purely sequential programme path. The `ArrayList` class, in all other respects equivalent to the `Vector` class, is not `synchronized`.

5.3 Quality Assurance

5.3.1 Software-Oriented Quality: Refactoring

The process of code refactoring presented in **??** aims at improving the *intrinsic* software quality. It has been applied to most classes, leading to more *simple*, *readable* and *maintainable* code. Those classes that have been subject to refactoring were fitted with individual unit tests so as to minimise the likelyhood of 'breaking' working code in the process.

5.3.2 Functionality-Oriented Quality

This aspect of quality is addressed by individual class tests (unit tests), class collaboration and environment integration tests (integration tests) and customer acceptance tests.

Unit Testing Unit testing aims at the assessment of the conformance of the each class, or group of classes, with a specified role in the design. This may be achieved by using *assertions* to test *pre* and *post-conditions*.
At first, the `run.ksh` script was used to launch test implementations in the `main()` method of each class. I then discovered the *JUnit* testing framework, that was consecutively used in a few instances.

Integration Testing Integration testing aims at the validation of the class collaboration model and of the system's architecture. It is not uncommon to find out at this (late) stage that the system's parts do not work in a satisfactory fashion together. For this reason the XP methodology (see **??**) recommends that daily integration tests be undertaken.
As already outlined, the process followed during the development phase involved building a fully tested and integrated—although partially functional—*daily* release. For instance, the first development stage was to create a fully tested system *skeleton*, with a client module capable of sending a fixed HTTP request message and a server module able to echo the message back to the client, thanks to the *NullPort* class. The non-trivial architecture, involving both subsystems, the application server, and the servlet engine was thus validated. In the next release, SOAP message generation was added, followed by SOAP interpretation in yet another release, etc. At a stage, the dummy `NullPort` class was replaced by the `Port` class to support actual Web services.

Acceptance Testing Acceptance testing is essentially performed by, or on behalf of the customer. The system specifications may be used for the purpose of assessing functional and non-functional requirements against the actual system. In the context of this project, two levels of acceptance tests were and will be carried out: a *systematic* verification against the relevant W3C specifications, and an *empiric* demonstration to the customer (i.e. project supervisor).

5.3.3 Documentation Quality

Because software documentation generation is part of the build process, it is guaranteed to be consistent with the system actually implemented.

5.4 Software Documentation

Java offers a standard way of creating code-level documentation using the `javadoc` utility. All the APIs—including the Java core APIs—are documented in this fashion. The Javadoc tool is part of the Java SDK, standard edition. Provided that the two following rules are respected, the utility will generate a set of HTML pages, organised hierarchically—laid out consistently with the core Java API documentation from Sun. First Javadoc comments must be enclosed in `/** ... */`, rather than in the "C-style" `/* ... */`. Then comments must be positioned immediately above the declaration of the elements of the language (classes, methods, fields) they describe. The `javadoc` utility will then be able to obtain semantic information from the comments and relate it to the syntactic information extracted from the main java code.

This approach is conceptually close to Knuth's *Literate Programming* initiative (see [Knuth 1984]). If software is more often than non considered to be poorly documented—Javadoc can help by providing a means to create consistent-looking, API-level documentation (i.e. the appropriate level of abstraction for the typical use of a middleware framework like XMLbroker)—with a small overhead for the programmer.

Figure 5.1 represents the documentation for the `Message` class. In the current version of Javadoc, comments are written in HTML, which unfortunately does introduce a strong inter-dependency between the content and the presentation-specific information.

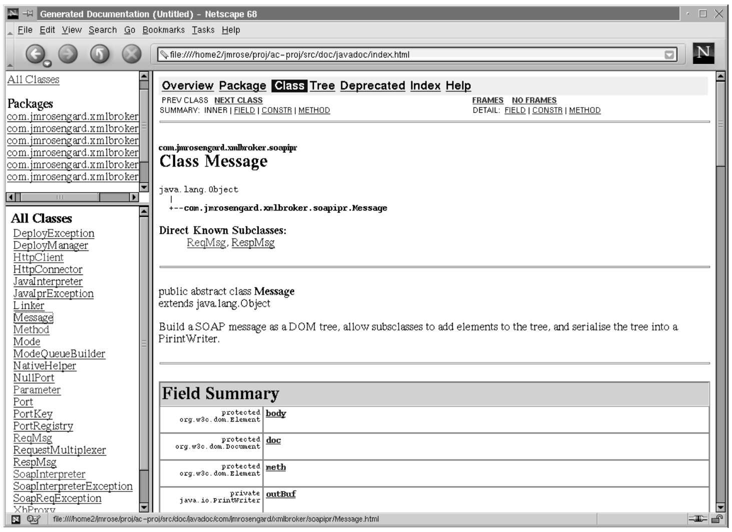

Figure 5.1: Javadoc HTML documentation

Bibliography

Arnold, K. & Gosling, J. [1998], *The Java^{TM} Programming Language*, second edn, Addison–Wesley.

Box, D., Ehenebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S. & Winer, D. [2000], 'Simple Object Access Protocol (SOAP) 1.1', W3C Note.
*http://www.w3.org/TR/SOAP

Donsez, D., Jean, S., Lecomte, S. & Thomas, O. [2001], Turning multi-application smart card services available from anywhere at anytime: A SOAP/MOM approach in the context of Java cards, *in* I. Attali & T. Jensen, eds, 'Proceedings of the Smart Card Programming and Security Conference', Vol. 2140 of *Lecture Notes in Computer Science*, Springer–Verlag, Cannes, France, pp. 83–94.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. [1995], *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison–Wesley.

Hunt, J. & McManus, A. [1998], *Key Java: Advanced Tips and Techniques*, Springer–Verlag.

IBM Research [2000], 'Java performance', *IBM Systems Journal* **39**(1).

Knuth, D. E. [1984], 'Literate programming', *IEEE Computer Journal* **27**(2), 97–111.

Prechelt, L. [2000], 'An empirical comparison of seven programming languages', *IEEE Computer Journal* **33**(10), 23–29.

Sametinger, J. [1997], *Software Engineering with Reusable Components*, Springer–Verlag.

Appendix A

Demonstration Service and Application

We present below a demonstration application using XMLbroker. Because the aim of the project is not to create a distributed application, but to engineer a middleware platform, the sole purpose of this trivial example is to demonstrate that the platform discussed in the past chapters works effectively.

A.1 Application Specification

The server hosting XMLbroker makes use of the Network Time Protocol (NTP) to update its internal time stamp every few minutes. This gives us enough credibility to establish a Web time service. For each SOAP request, the server will simply return a timestamp.

A.2 Implementation

A.2.1 Service Definition

The service is defined by the following service interface. The method `getNtpTime()` outputs a single parameter, as a character string. `XbString` is a *mutable* version of the Java `String`. As already discussed, the interface is tagged as a service interface by extending the `XbService` interface.

```
package deployed;
public interface NtpService extends com.jmrosengard.xmlbroker.deploy.XbService
 public void getNtpTime__out (com.jmrosengard.xmlbroker.deploy.XbString time);
}
```

Code Listing A.1: Service interface for the demonstration service and application

A.2.2 Service Implementation

The actual implementation of the service is trivial. The current time is converted to a string and passed to the output parameter.

```
public void getNtpTime(XbString time){
time.set((new java.util.Date()).toString());
}
```

Code Listing A.2: Demonstration service implementation

A.2.3 Service Integration

The following Java code implements the client application, and integrate the remote service. The startup shell script defines the server host, service name and service interface.

```
// instantiate dynamic proxy
XbProxy proxy=new XbProxy();
// create ArrayList for 'in' and 'out' parameters
ArrayList ins=new ArrayList();
ArrayList outs;
// invoke service
outs=proxy.invokeService(new ArrayList());
// display result
System.out.println("The time returned by server\n"
 +System.getProperty("xmlbroker.serverHost")
 +" is:\n "
 +((Parameter)outs.get(0)).getValue().toString());
```

Code Listing A.3: Demonstration service integration in the client application

```
#!/bin/sh
#
# Run XMLbroker demo
#
ip=`rtsrp`
prop="-Dxmlbroker.serviceName=http://www.jmrosengard.com/xmlbroker#demo
      -Dxmlbroker.serverHost=http://$ip:8080/xmlbroker/xb1
      -Dxmlbroker.serviceInterface=deployed.NtpService"
cp="-classpath XMLbroker.jar:$CLASSPATH"

java $prop $cp NtpDemo
```

Code Listing A.4: Demonstration application startup script

A.3 Sample SOAP messages

```
POST /xmlbroker/xb1 HTTP/1.1
Content-Type: text/xml; charset="UNICODE-1-1-UTF-8"
Content-Length: 326
SOAPAction: http://www.jmrosengard.com/xmlbroker#demo


<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<demo:GetNtpTime xmlns:demo="http://www.jmrosengard.com/xmlbroker/demo">
<!-- no input parameter -->
</demo:GetNtpTime>
</SOAP-ENV:Body>
```

```
</SOAP-ENV:Envelope>
```

Code Listing A.5: SOAP request message for the demonstration application

```
1  HTTP/1.1 200 OK
2  Content-Type: text/xml
3
4
5  <SOAP-ENV:Envelope
6  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
7  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
8  <SOAP-ENV:Body>
9  <demo:GetNtpTimeResponse
10 xmlns:demo="http://www.jmrosengard.com/xmlbroker/demo">
11 <demo:time>Sun Feb 17 18:10:33 GMT 2002</demo:time>
12 </demo:GetNtpTimeResponse>
13 </SOAP-ENV:Body>
14 </SOAP-ENV:Envelope>
```

Code Listing A.6: SOAP response message for the demonstration application

Note line 9 the SOAP naming convension. For a method element `GetNtpTime` in the request, the element name in the response is `GetNtpTimeResponse`.

Appendix B

UML Artefacts

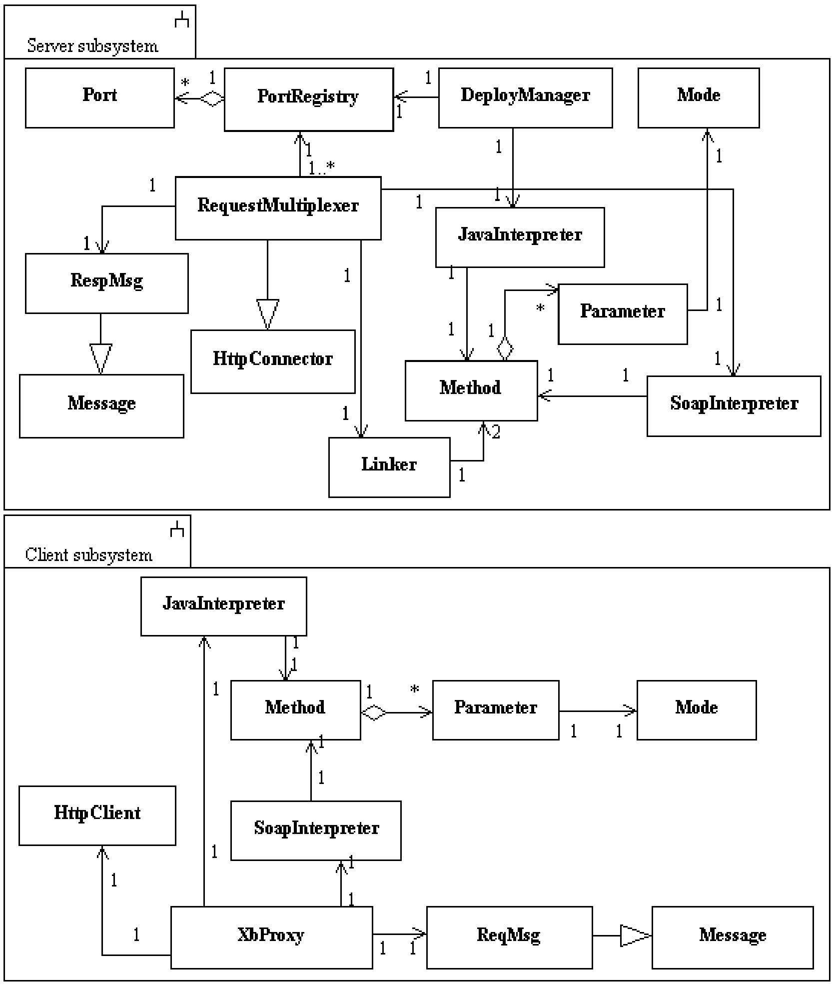

Figure B.1: Analysis class models for the client and server subsystems

Package name	Package role
broker	server subsystem
client	client subsystem
deploy	Web service deployment
javaipr	Java interpeter
os	C helper functions, Java wrappers
soapipr	SOAP interpreter and message builder

Table B.1: Distribution of roles by package

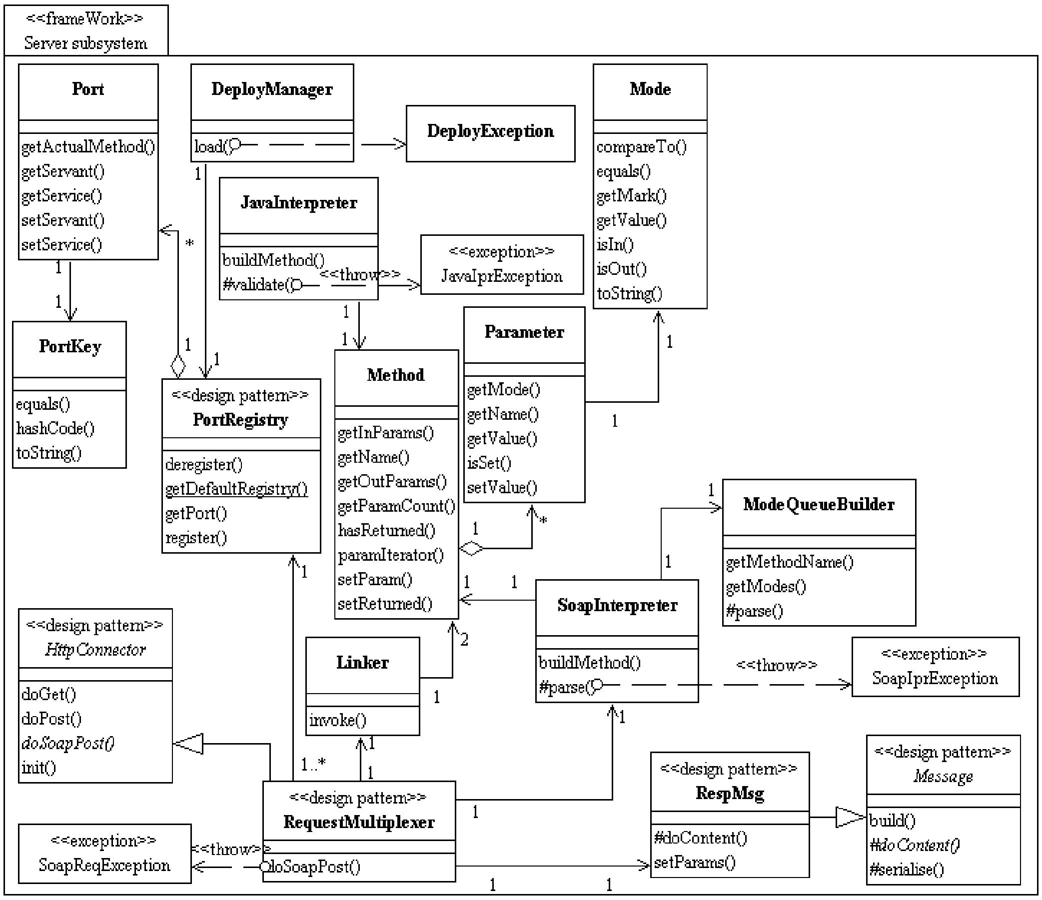

Figure B.2: Design class model for the server subsystem. To avoid overloading the diagram, methods are assumed public by default. Fields are not represented as they are not essential to the comprehension of the class model.

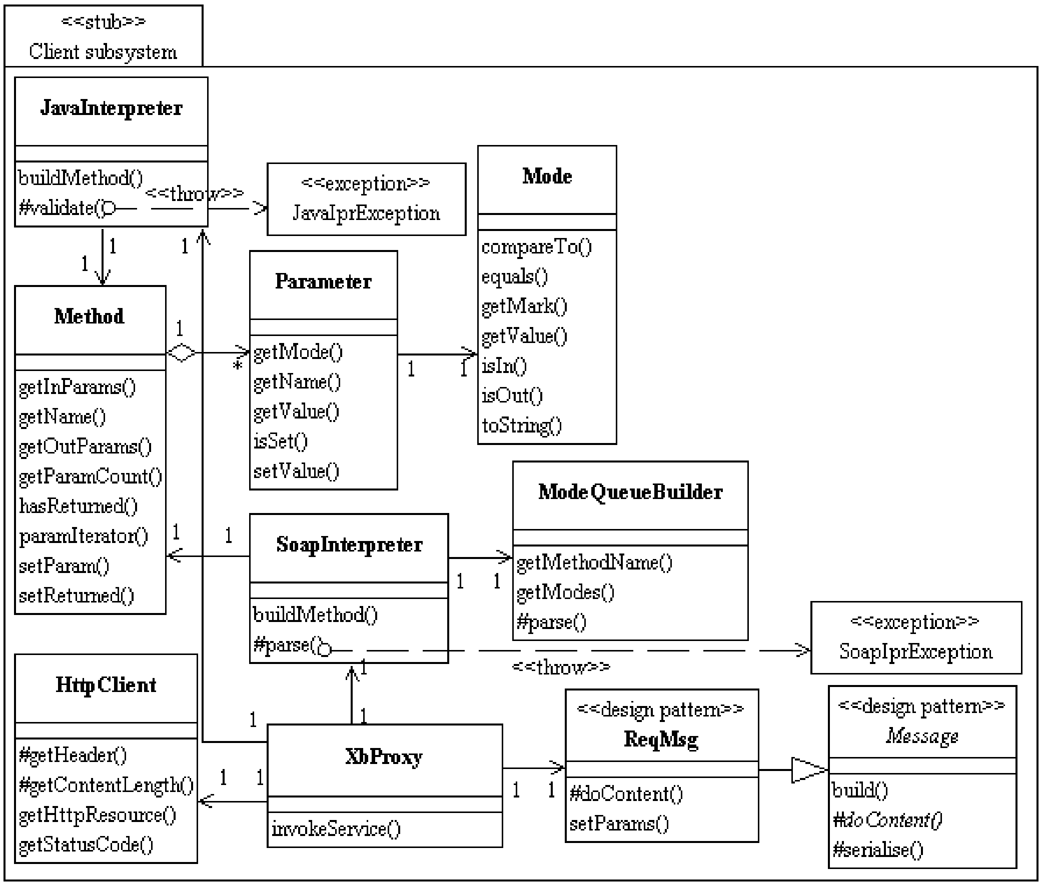

Figure B.3: Design class model for the client subsystem. To avoid overloading the diagram, methods are assumed public by default. Fields are not represented as they are not essential to the comprehension of the class model.

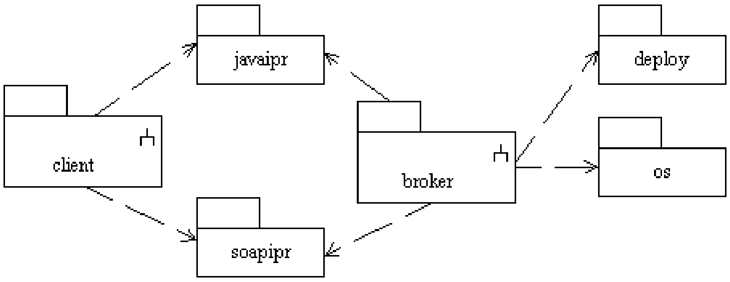

Figure B.4: Logical package structure of XMLbroker