

# NEOOM: A Web and Object Oriented Middleware System

Pasqualino 'Titto' Assini (titto@nesstar.com)  
FASTER Project

November 9, 2001

# Contents

<b>1</b>	<b>Introduction</b>	<b>8</b>
1.1	Recent Changes and Additions . . . . .	9
1.2	Structure of the Document . . . . .	9
1.3	History of the Project . . . . .	10
1.4	The promise of the Object Web . . . . .	10
1.5	Why an OO Web Would Be a Better Web . . . . .	12
1.6	Requirements . . . . .	12
1.7	NEOOM: A Web and Object Oriented Middleware System . . . . .	13
1.8	Overall Operation . . . . .	13
1.9	Related Work . . . . .	15
1.9.1	Service Specification . . . . .	17
1.9.2	Service Invocation . . . . .	17
1.9.3	Service Discovery . . . . .	18
1.9.4	Service Composition . . . . .	19
1.9.5	Summary . . . . .	20
1.10	Future Work . . . . .	20
<b>2</b>	<b>Object Model</b>	<b>21</b>
2.1	The NEOOM Object Model . . . . .	21
2.2	Type System . . . . .	22
2.3	Properties . . . . .	24
2.4	Methods . . . . .	25
2.4.1	Parameters . . . . .	26
2.4.2	Exceptions . . . . .	27
<b>3</b>	<b>Network Protocol</b>	<b>28</b>
3.1	Requirements . . . . .	29
3.2	Design Options and Related Work . . . . .	29
3.2.1	Evaluation . . . . .	30
3.2.2	Supporting Multiple Network Protocols . . . . .	31
3.3	Accessing an Object State . . . . .	31
3.3.1	Accessing a Single Property . . . . .	31
3.3.2	Accessing the Complete Object State . . . . .	32

3.3.3	Extended/Reduced Information Transfer . . . . .	33
3.3.4	Accessing a NEOOM Class Definition . . . . .	33
3.3.5	Caching . . . . .	33
3.4	Method Calls . . . . .	34
3.5	Exceptions . . . . .	36
3.6	Modifying an Object State . . . . .	36
3.7	URL Representation of Method Calls . . . . .	37
3.8	Access Control Protocol . . . . .	38
3.8.1	Tracking User Identity . . . . .	39
3.9	$\pi$ -Calculus Protocol Specification . . . . .	39
<b>4</b>	<b>Java Implementation</b>	<b>43</b>
4.1	Introduction . . . . .	44
4.2	Object Server . . . . .	44
4.2.1	Requirements . . . . .	44
4.2.2	Overall Operation . . . . .	44
4.3	Access Control Unit . . . . .	45
4.3.1	Requirements . . . . .	45
4.3.2	Overall Operation . . . . .	45
4.3.3	Conditions . . . . .	48
4.3.4	Users . . . . .	48
4.3.5	Actions . . . . .	48
4.4	Object Browser . . . . .	49
4.4.1	Requirements . . . . .	49
4.4.2	Overall Operation . . . . .	49
4.5	APIMaker . . . . .	50
4.5.1	Requirements . . . . .	50
4.5.2	Overall Operation . . . . .	50
4.5.3	Format of the Java Client Stubs . . . . .	50
4.5.4	Format of the Java Server Skeletons . . . . .	50
4.5.5	Synopsis . . . . .	54
4.6	Faster Object Server . . . . .	55
4.6.1	Requirements . . . . .	55
4.6.2	Design Options . . . . .	56
4.6.3	Overall Operation . . . . .	58
4.6.4	Mapping between NEOOM objects and EJB objects . . . . .	58
4.6.5	NEOOM HTTP Protocol Support . . . . .	58
4.6.6	ACU Support . . . . .	58
4.6.7	Development Guide . . . . .	58
4.7	Future Work . . . . .	59
4.7.1	XMI mapping and Argo/UML support . . . . .	59
4.7.2	Secure Socket Layer Support . . . . .	59

<b>A</b>	<b>Source Code</b>	<b>60</b>
A.1	RDF Interfaces . . . . .	60
A.2	Java Code . . . . .	65
A.2.1	Server . . . . .	65
A.2.2	Access Control Unit . . . . .	67
A.2.3	API Maker . . . . .	70
A.2.4	Object Browser . . . . .	73

# List of Figures

2.1	NEOOM Object Model . . . . .	23
3.1	Protocol UML Sequence Diagram . . . . .	40
4.1	Server UML Class Diagram . . . . .	46
4.2	ACU UML Class Diagram . . . . .	47
4.3	Object Browser User Interface . . . . .	51
4.4	Object Browser UML Class Diagram . . . . .	52
4.5	ApiMaker UML Class Diagram . . . . .	53
4.6	Faster Server design options . . . . .	57

# List of Tables

1.1	Web Services Functions and Technologies . . . . .	16
2.1	Object Model Namespaces . . . . .	22
2.2	NEOOM primitive types . . . . .	24
2.3	rdf:Property additional properties . . . . .	25
2.4	n:Method methods . . . . .	25
2.5	n:Method properties . . . . .	26
2.6	n:Parameter properties . . . . .	26
2.6	n:Parameter properties . . . . .	27
3.1	FORM INPUT and MIME types . . . . .	35
3.1	FORM INPUT and MIME types . . . . .	36
4.1	Object Browser CGI Parameters . . . . .	49
4.2	APIMaker Synopsis . . . . .	54
A.1	Java Libraries . . . . .	65

# List of Abbreviations

<b>ACU</b>	Access Control Unit
<b>CGI</b>	Common Gateway Interface
<b>COM+</b>	Component Object Model +
<b>CORBA</b>	Common Object Request Broker Architecture
<b>DAML-S</b>	DAML Web Service Ontology
<b>EJB</b>	Enterprise Java Beans
<b>HTTP</b>	Hypertext Transfer Protocol
<b>IDL</b>	Interface Definition Language
<b>IIOP</b>	Internet Inter-ORB Protocol
<b>JAAS</b>	Java Authentication and Authorization Service
<b>JAR</b>	Java Archive
<b>JDO</b>	Java Data Objects
<b>JNDI</b>	Java Naming and Directory Interface
<b>NEOOM</b>	NEsstar Object Oriented Middleware
<b>OO</b>	object-oriented
<b>OOL</b>	Object Oriented Language
<b>OOM</b>	object-oriented middleware
<b>OOW</b>	Object-Oriented Web
<b>RMI</b>	Remote Method Invocation
<b>RPC</b>	Remote Procedure Call
<b>SSL</b>	Secure Socket Layer

**UA** User Agent

**UDDI** Universal Description, Discovery and Integration

**UML** Unified Modeling Language

**W3C** World Wide Web Consortium

**WIDL** Web Interface Definition Language

**WSDL** Web Services Description Language

**WSFL** Web Services Flow Language

**WWW** World Wide Web

**XMI** XML Metadata Interchange

# Chapter 1

## Introduction

### Contents

---

<b>1.1</b>	<b>Recent Changes and Additions . . . . .</b>	<b>9</b>
<b>1.2</b>	<b>Structure of the Document . . . . .</b>	<b>9</b>
<b>1.3</b>	<b>History of the Project . . . . .</b>	<b>10</b>
<b>1.4</b>	<b>The promise of the Object Web . . . . .</b>	<b>10</b>
<b>1.5</b>	<b>Why an OO Web Would Be a Better Web . . .</b>	<b>12</b>
<b>1.6</b>	<b>Requirements . . . . .</b>	<b>12</b>
<b>1.7</b>	<b>NEOOM: A Web and Object Oriented Middle- ware System . . . . .</b>	<b>13</b>
<b>1.8</b>	<b>Overall Operation . . . . .</b>	<b>13</b>
<b>1.9</b>	<b>Related Work . . . . .</b>	<b>15</b>
1.9.1	Service Specification . . . . .	17
1.9.2	Service Invocation . . . . .	17
1.9.3	Service Discovery . . . . .	18
1.9.4	Service Composition . . . . .	19
1.9.5	Summary . . . . .	20
<b>1.10</b>	<b>Future Work . . . . .</b>	<b>20</b>

---

### Abstract

Although the advantages of object-oriented programming are well known none among the main distributed object-oriented middlewares (COM+, CORBA and Enterprise Java Beans) has been widely adopted for the developing of Internet applications. Developers seem to perceive these approaches as over-complex (COM+ and CORBA), proprietary (COM+ and EJB), and incompatible with current World Wide Web development practices. Recognizing these difficulties more WWW-friendly proposals based on XML are starting to appear (WSDL,DAML-S, etc.). This report describes the

NEsstar Object Oriented Middleware (NEOOM) system. NEOOM aims at bridging the gap between heavyweight object-oriented middleware systems and the Web. It's a lightweight Web-friendly object-oriented middleware that can provide most of the advantages of object-orientation while maintaining full compatibility with the existing WWW infrastructure and requiring only modest changes to existing applications to allow them to participate to the Object Web. To ensure compatibility with the WWW it uses an extension of the Resource Description Framework (a World Wide Web Consortium standard) as its Interface Definition Language and the Hypertext Transfer Protocol as its network protocol.

**Keywords:** Web Services, Web objects, RDF, distributed systems

## 1.1 Recent Changes and Additions

Sections modified or added in the last revisions of this report.

Revision 2001/10/30:

- 4.6

Revision 2001/10/08:

- 4.7.2

Revision 2001/09/21:

- 1.9.4
- 4.7

## 1.2 Structure of the Document

This report contains a specification of the NEsstar Object Oriented Middleware (NEOOM) system and a description of its implementation in the Java programming language.

This initial chapter is meant to provide an informal but complete introduction to the system. It contains an explanation of the problem that NEOOM was developed to solve, its functional requirements, the history of its development, a description of its overall operation and a comparison with similar systems (a more specific comparison of different network protocols can be found in section 3.2).

A (semi-)formal definition of the system is provided in the Object Model (2) and Network Protocol (3) chapters. The Java implementation is described in chapter 4. In some cases the behaviour of the Java implementation differs slightly from the specification. A set of *Java implementation notes* is provided to document these discrepancies.

Some indications on possible future evolutions of the system are drawn in section 1.10.

The source code (??) and the bibliography are in the Appendices.

The latest version of this report is available (in PDF format) at:

<http://www.nesstar.org/sdk/neoom.pdf>.

### 1.3 History of the Project

The idea of an RDF-based object-oriented middleware has initially appeared in the design of the client/server protocol of the EU-funded project NESSTAR [NESa]. The work has continued during the EU-funded project FASTER [FAS]. A first implementation of the FASTER version of NEOOM has been completed at the beginning of year 2000. Some preliminary results have been presented at an EU Semantic Web Technologies Workshop [Ass00] in November 2000 and with a poster at the WWW10 conference [Ass01] in May 2001.

### 1.4 The promise of the Object Web

The imminent advent of the object-oriented (OO) Web has been announced many times in the last few years. The advantages of OO programming with respect to more traditional approaches are so evident that most observers had assumed that it was only a question of time before objects would become a basic feature of Internet development. As early as 1996 Marc Andreessen [And96] could confidently affirm that:

*The next shift catalyzed by the Web will be the adoption of enterprise systems based on distributed objects and Internet Inter-ORB Protocol (IIOP) ... The next version of Netscape Navigator and Netscape SuiteSpot Servers will be IIOP-compliant. IIOP will be integral to everything we do. Netscape Navigator will be able not only to browse content but also to browse objects. We expect to distribute 20 million IIOP clients over the next 12 months and millions of IIOP-based servers over the next couple of years. We'll put the platform out there so that people can start developing for it.*

While Netscape, Sun and other companies provided the basic tools the idea that the Object Web was just behind the corner had been widely popularized by the extremely successful books on this subject by Robert Orfali et al. [OHE97]. The main question for most observers was not if the OO Web was going to happen but rather if it would have been dominated by the Common Object Request Broker Architecture (CORBA) [COR99] or by Microsoft's Component Object Model + (COM+) [COM]. Unfortunately after such a brilliant start the Object Web has not made much progress. Even if highly sophisticated object-oriented middleware systems such as CORBA, COM+ and Enterprise Java Beans (EJB) [EJB] have been available for years

most of current distributed systems are still built using relatively "primitive" Web techniques such as Common Gateway Interface (CGI) scripts [CGI] and Java Servlets [SER].

The Object Web is simply not happening: a simple delay or a major failure?

It's not easy to see why such a promising technology might have failed to capture the developers' imagination but some of these elements might have contributed:

- There is a considerable conceptual and technical mismatch between object-oriented middleware (OOM) and World Wide Web (WWW) development techniques
- OOMs are perceived by most developers as complex and overkill for most applications
- A growing number of intranets are protected by firewalls that, almost invariably, will allow only email and Hypertext Transfer Protocol (HTTP) traffic. This is bad news for OOMs that are based on non-HTTP protocols.<sup>1</sup>
- The conflict between CORBA and COM+ has confused many developers

Both CORBA and COM+ originate in the pre-Internet era and lack two essential characteristics:

- Simplicity
- Compatibility with existing WWW technologies

Simplicity is probably nowhere as important as in the Internet environment. A good example is given by the WWW itself, an hypertext system that provides only the most basic form of hypertext linkage: the, often broken, uni-directional link. At the beginning of the Nineties, when the WWW was born, there were a number of much more sophisticated hypertext systems available but none of them has had an impact even remotely comparable to that of the "humble" WWW. We now live in a world that has been thoroughly changed by the effects of the marriage of the Internet with such a simple technology. OO development has the potential of propelling the Web to a new era but probably needs to go through a similar process of simplification.

---

<sup>1</sup>This problem can be solved by using HTTP-tunnelling, that's to say by mapping the OOM wire protocol on top of HTTP. Unfortunately this adds extra complexity, it's not standardized and doesn't work particularly well as the unidirectional nature of the HTTP protocol (a typical Request-Reply protocol) is not a good match for OOM protocols that normally assume bidirectional communication among the objects.

## 1.5 Why an OO Web Would Be a Better Web

There are two basic differences between the current "procedural" web and the Object-Oriented Web (OOW):

- the interfaces of the OOW are explicitly stated
- the interfaces are typed

Explicit and typed interfaces can go a long way in simplifying WWW development and improving the WWW surfing experience.

For developers they mean:

- accurate documentation, especially important now that B2B commerce requires the integration of separately developed WWW systems
- availability of generic tools (for ex: object browsers) to explore, develop, debug and integrate distributed systems

For users the OOW would in particular mean browsers that can make sense of a more significant part of the information stored in the WWW and that could use this increased understanding to be more helpful. Consider for example an user that is looking for a person. He knows that this person works for some university but doesn't know which one. Most university have a search facility on their WWW sites. A smart browser would be able to apply the user query in parallel to a number of different university sites. Unfortunately where a human user sees an academic WWW site with a search service a browser sees only another HTML page with a form to display. It has no way of knowing the 'meaning' of what it's displaying.

If, on the contrary, the search services of the different sites were defined as objects of the same type the user browser could easily recognize this fact and offer the user the possibility of searching them in parallel.

The OOW is actually a particular instance of a much wider vision: that of the Semantic Web. An excellent introduction to this concept and to its potential impact on the way we access to distributed information has recently appeared in Scientific American in an article by the inventor of the WWW, Tim Berners-Lee [HBL01].

## 1.6 Requirements

To realize the promise of the Object Web we need a form of middleware that can bridge the gap between heavyweight object-oriented middleware systems and the WWW. It has to be able to provide most of the advantages of object-orientation while maintaining full compatibility with the existing WWW

infrastructure and requiring only modest changes to existing applications to allow them to participate to the Object Web.

In order to do so it must:

- have a simple OO model, possibly based on existing standards
- integrate easily with the normal web, it must be for example be possible to start the execution of a an OO method call by clicking on an hypertext link
- use a network protocol compatible with HTTP

As the WWW is an open environment it has also to be able to protect its services by unauthorized access. This leads to another requirement:

- must provide access control (at the method and property level)

## 1.7 NEOOM: A Web and Object Oriented Middleware System

The basic characteristics of NEOOM closely reflects these requirements:

- the Object Model is an extension of the OO model defined by the World Wide Web Consortium (W3C) RDF and RDF Schema standards
- remote objects are accessed through an HTTP-based network protocol
- normal web browsers can execute any operation on NEOOM objects
- has a flexible access control system

## 1.8 Overall Operation

We will now briefly examine how NEOOM works from two points of view: that of an application programmer that wants to use some NEOOM object and that of an object developer that wants to create new NEOOM objects.

NEOOM is a system to specify, create and "publish" objects. An object is an entity that carries some information and that can execute operations. There is a similar architecture that we are all quite familiar with: the World Wide Web. The Web is a system that allows us to create entities, such as documents or images, and publish them easily. Each entity has an URL and if we know the URL of a particular entity we can "point" our web browser to it and examine it.

The web is not composed only by static entities, it also offers services. We can use the web to perform operations such as buying a flight ticket or

searching a library catalogue. The way we perform these operations is by filling in forms embedded in HTML pages.

NEOOM Objects are very similar to web documents. They "live" at a given URL and can be accessed by entering their URL in a web browser. What we get back when we access a NEOOM object is a description of the object state (that's to say the value of its properties) coded in RDF. Something that looks like this:

```
<r:RDF xmlns:r="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:n4="http://www.nesstar.org/rdf/Server#"
  xmlns:n="http://www.nesstar.org/rdf/"
  xmlns:n5="http://www.nesstar.org/rdf/Catalog#"
  xmlns:n3="http://www.nesstar.org/rdf/Dataset#"
  xmlns:s="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#">
<n:Server r:about="http://155.245.254.82:4000/obj/Server">
<n4:version>1.1</n4:version>
<n4:revision>Thu Nov 09 19:21:34 GMT+00:00 2000</n4:revision>
<n4:debug>off</n4:debug>
<n4:services r:resource="http://155.245.254.82:4000/obj/services" />
<n:administrators r:resource="http://155.245.254.82:4000/obj/administrators" />
<s:comment>The UKDADEV Server.</s:comment>
<n:accessCondition r:resource="http://155.245.254.82:4000/obj/serverCondition" />
<s:label>UKDADEV</s:label>
</n:Server>
</r:RDF>
```

Try to access an online NEOOM object:

<http://nesstar.data-archive.ac.uk/obj/Server>

The most important property of a NEOOM object is its type (or class). A NEOOM class is just another object and so it can also be accessed with a normal browser. What we get back when we access a NEOOM class is a description of all the properties and methods (the operations) of the objects that are instances of the class.

Try to access an online NEOOM class:

<http://www.nesstar.org/rdf/Server/>

So by simply accessing two ordinary URLs we can find out what are the values of the properties of an object and what kind of functionality it offers. If we want to perform one of the object operations we can do it in the same way as we would execute any other normal web operation: by filling in a web form and sending it to the object URL.

The RDF description of a NEOOM object is human readable and so is the definition of its methods but it is convenient to have a tool that makes this information easier to read and that can build for us the HTML forms that are needed to execute the object operations. One such tool is the Object

Browser (see the section 4.4 for a description). The Object Browser is an universal client that can be used to provide a simple HTML interface to any NEOOM object. For a snapshot of its interface see figure 4.3.

Try the Object Browser online:

<http://nesstar.data-archive.ac.uk/browser?url=/obj/Server>

So, if we are a web developer interested in using some service (operation) offered by a NEOOM object we can use the Object Browser to examine the object and then cut and paste the forms that correspond to the operations that we want to apply into our web pages or, for more sophisticated applications, we can use the HTTP library of our preferred programming language to perform an operation equivalent to the submission of the form.

What if we want to create a new class of NEOOM objects? The development process is similar to that of traditional middleware systems (such as CORBA). We would start by specifying the class features (properties and methods) in RDF. If we intend to use Java to implement our object we would then use the APIMaker tool (see section 4.5) to generate the client stubs and the server skeletons. The server skeletons are a set of classes that we have to extend to implement the operations of our class. We can then pack the skeletons plus our implementation in a Java Archive (JAR) file and upload it to a NEOOM server (see section 4.2). We can now create some objects in the server, maybe by writing them down in RDF in the server configuration file or by instructing the server to retrieve them from a database. The final step consists in connecting to our newly created objects either by using an universal client such as the Object Browser or programmatically using the client stubs generated by APIMaker.

## 1.9 Related Work

The *impasse* of the Object Web and the necessity of more WWW-friendly middleware standards have been apparent for a while. Some proposals aimed at solving this problem such as WebBroker [TL98] and the Web Interface Definition Language (WIDL) [Mer97] have been published as W3C Notes as early as 1997 (for a good review of WIDL and other similar proposals see [Man98], for a more recent and shorter version see [Man]).

But only very recently there has been an upsurge in interest, largely due to Microsoft's .Net initiative [MNE], and new proposed standards, backed by major software houses or governmental agencies, have started to appear. The label "Web Services" under which these new technologies are usually classified express quite well the intention of marrying the world of WWW development with that of traditional OOMs and Remote Procedure Call (RPC) systems.

In this section I will mainly focus on two of these proposals that are particular significant with respect to NEOOM: the Web Services Description

Language (WSDL) [Chr01] and the DAML Web Service Ontology (DAML-S) [DAM01].

WSDL is significant from a practical point of view as it has the backing of some major players of the IT world such as Microsoft and IBM. DAML-S is technically significant as, just as NEOOM, is based on (or at least expressed in) the RDF language.

Another major standard is ebXML. I won't discuss it as it is more geared to business transactions than to generic web services systems. In the words of its creators, ebXML:

*sponsored by UN/CEFACT and OASIS, is a modular suite of specifications that enables enterprises of any size and in any geographical location to conduct business over the Internet. Using ebXML, companies now have a standard method to exchange business messages, conduct trading relationships, communicate data in common terms and define and register business processes [ebX].*

Another important standard is the Universal Description, Discovery and Integration (UDDI) [UDD], a distributed registry of web services that we will briefly examine in section 1.9.3.

But what is exactly a Web Service? According to Microsoft:

*A Web Service is programmable application logic accessible using standard Internet protocols. [MNE]*

This is a very generic definition, a more restrictive one might be that Web Services are XML-based, distributed, object-oriented (at least to some degree) middleware frameworks designed to integrate easily with the Web.

To get a better understanding of the relative position of these technologies is useful to classify them with respect to some of the major functions of a Web Services system:

<b>Function</b>	<b>Technologies</b>
Specification	ebXML, WSDL, DAML-S, NEOOM
Execution	ebXML, WSDL, DAML-S, NEOOM
Discovery	ebXML, UDDI
Composition	DAML-S

Table 1.1: Web Services Functions and Technologies

### 1.9.1 Service Specification

WSDL, DAML-S and NEOOM are mainly Web Service specification languages.

WSDL uses the following elements to define a web service [Chr01, sect.2.1]:

**Message** an abstract definition of the data being transmitted. A message consists of logical parts, each of which is associated with a definition within some type system.

**Operation** an abstract description of an action supported by a service. Each operation refers to an input message and output messages.

**Port Type** an abstract set of operations supported by one or more endpoints.

**Binding** a concrete protocol and data format specification for a particular port type.

**Port** a single endpoint defined as a combination of a binding and a network address.

**Service** a collection of related endpoints.

In more traditional RPC terminology *messages* would be *function signatures*, *message parts* would be *parameters* and a *portType* an *interface definition*. WSDL has no separate concept of the state of a service/object, only methods can be defined (naturally methods can be used to provide access to the service state).

DAML-S describes services through a *ServiceProfile*. For each service is possible to specify its *inputs*, *outputs*, *preconditions* and *effects*. Preconditions are: “conditions that should be satisfied prior to the service being requested”, effects are “events that are caused by the successful execution of a service” [DAM01, sect.4.2]. It’s also possible to specify *functional attributes* such as the geographical scope of the service, the quality guarantees offered, etc. [DAM01, sect.4.3].

NEOOM way of describing services is conceptually very similar to that of traditional IDLs such as the CORBA IDL. What are described are objects, and what is specified are the object properties and methods.

### 1.9.2 Service Invocation

To be able to actually use a web service it must be accessible through some network protocol. WSDL offers a lot of flexibility in this area. The standard defines bindings to HTTP GET/POST, MIME and SOAP and can be extended to support additional protocol bindings.

DAML-S provides bindings through *Service Grounding* objects. This part of the ontology hasn't been defined yet but it is likely that it will provide functionality similar to WSDL.

NEOOM defines a standard binding to HTTP and allows for the addition of further protocols (see section 3.2.2).

### 1.9.3 Service Discovery

As the huge success of the web search engines has demonstrated the biggest problem in a distributed environment is how to find out what is available. The same problem applies to the Object/Semantic Web: how to find out what services are available?

UDDI tries to solve this problem by providing:

*a group of web-based registries that expose information about a business or other entity and its technical interfaces (or API's). These registries are run by multiple operator sites, and can be used by any business that wants to make their information available, as well as anyone that wants to find that information [UDD]*

UDDI is both a network of servers that store information on business entities and the services that they offer and the specification of an API to query the network. UDDI offers basically three services [Col]:

**White Pages** that hold basic information regarding business entities (name, description, contact info, known identifiers, etc)

**Yellow Pages** that order business entities by category on the base of a set of taxonomies (geographical, by sector of activity, etc.)

**Green Pages** that hold technical information on the kind of web services offered by business entities together with the information necessary to bind to these services automatically

Microsoft is also working on a simpler standard called Discovery Protocol (Disco). The specification is not yet available but it will define a document format (based on XML) to describe services and a protocol to access them. This will be useful for example in Intranets or in networks of business partners that are already aware of each other.

WSDL, DAML-S and NEOOM do not offer specific service discovery functionality. WSDL will use UDDI for this function. DAML-S and NEOOM could also use UDDI or similar services.

NEOOM objects can be accessed as normal Web document so they could also be indexed by traditional WWW search engines. They can also be "discovered" by following links from related services, just as one can discover

a new interesting web page by following a link from a known one. In the NESSTAR system, that is based on NEOOM, the user consults a web home page that contains a catalogue of available search services and can make his client aware of them by simply "clicking" on their links.

#### 1.9.4 Service Composition

Service composition is the creation of new services by connecting existing, simpler, services. DAML-S provide a set of *composition templates* [DAM01, sect.5] for sequential, parallel, conditional, etc. composition. The result looks rather like a "macro" language for services.

WSDL and NEOOM do not provide any such facility. The composition of WSDL or NEOOM services can naturally be operated by the User Agent or by specialised web services.

Some very simple examples of UA-side service composition are provided by the Nesstar Explorer [NESb]. The Nesstar Explorer can apply a search to multiple search services and aggregate the results. This is an example of a specific, hardwired, composition. Another form of composition is provided by the bookmark tree. Each bookmark corresponds to either a web resource or a NEOOM object-oriented call. Bookmarks are held into folders and folders can be instructed to execute the contained bookmarks, or folders, either sequentially or in parallel, once or a number of times. The Bookmark trees can be published as documents on the web or otherwise exchanged across clients. The NESSTAR Explorer bookmark tree is effectively a very simple visual 'macro' language that can be used for generic, but loose, composition of NEOOM and WWW services.

The difference between loose and hardwired composition of services lies in the knowledge that the composition mechanism has of the composed services. In the case of the multiple search the UA is fully aware of the semantic of the composed operation. On the contrary the bookmark tree knows only how to replay the bookmark but it's totally unaware of their meaning. Hardwired composition is powerful but has to be explicitly programmed. Loose composition is generic but weak. The most interesting area for composition lies in the middle of these two extremes: the objects/services provide sufficient information about themselves ("composition hints") to allow a composition mechanism, that is aware of the meaning of these hints but not of the specific object/services, to combine them, possibly automatically, in meaningful and powerful ways.

XLANG [Tha] and Web Services Flow Language (WSFL) are two recent proposals by Microsoft and IBM for web services compositions. They are expected to be merged and presented to the W3C for approval before the end of the year.

### 1.9.5 Summary

DAML-S is a very ambitious attempt at defining an extensive ontology for Web Services that includes aspects as varied as the service interface, its functionality, the composition with other services, etc. The extensiveness of DAML-S is both its attraction and its weak point. Creating DAML-S compliant services and clients might prove to be rather difficult and this might reduce the practical value of the system. As both NEOOM and DAML-S are expressed in RDF NEOOM systems might be able to incorporate some of the advanced features of DAML-S when their usefulness will have been proven.

WSDL weak point seems to be its RPC style. The lack of object-oriented features reduces its expressiveness and extensibility. Nevertheless the great commercial support that it enjoys will probably assure its success.

NEOOM is a simple, purely object-oriented, middleware system designed to integrate as easily as possible with the web. It offers less functionality than DAML-S or WSDL/UDDI but, being based on common WWW standards, it should be possible to integrate it with other systems to enrich its functionality.

No final conclusions can be drawn. All these proposals are still in an immature stage and are bound to evolve significantly. For Web services these are definitely very early times.

## 1.10 Future Work

There are many possible directions in which NEOOM might evolve. The most profitable path to useful improvements in both the specification and the implementation of the system seems to be that of the integration of NEOOM with other technologies.

At the specification level there is certainly much to be learned by similar web services description languages such as DAML-S. The integration with standards such as UDDI, that provide functionality not covered by NEOOM, might be even more beneficial. Extending the range of supported network protocols to include SOAP might improve interoperability with other OOM systems and lead to additional possibilities of reuse and integration.

Another interesting technology is the XML Metadata Interchange (XMI) standard. A mapping between XMI and the NEOOM object model would allow the usage of advanced UML design tools such as Argo/UML (and possibly its commercial variant Poseidon as well as similar UML case tools) for the creation of NEOOM models.

## Chapter 2

# Object Model

### Contents

---

<b>2.1</b>	<b>The NEOOM Object Model . . . . .</b>	<b>21</b>
<b>2.2</b>	<b>Type System . . . . .</b>	<b>22</b>
<b>2.3</b>	<b>Properties . . . . .</b>	<b>24</b>
<b>2.4</b>	<b>Methods . . . . .</b>	<b>25</b>
2.4.1	Parameters . . . . .	26
2.4.2	Exceptions . . . . .	27

---

## 2.1 The NEOOM Object Model

The NEOOM Object Model is an extension of the OO model defined by the W3C RDF [LS99] and RDF Schema [GB00] standards. RDF/RDFS provide a simple OO model that is quite suitable for the description of distributed system. The missing element is the possibility of describing *behaviour* (method calls in OO terminology). The NEOOM extensions add this missing feature transforming RDF in a fully-fledged Interface Definition Language (IDL).

The NEOOM Object Model is represented, as an Unified Modeling Language (UML) Class Diagram, in figure 2.1. The diagram shows clearly that the concepts that compose the model are defined in different XML namespaces [BHL99] (represented in the diagram as UML packages). The properties of each class are normally defined in the same namespace as the class. When this is not the case this is shown by prefixing the property name with the name of the namespace followed by an underscore. So for example the *about* property of the *rdfs:Resource* class is not in the *rdfs* namespace but rather in the *rdf* namespace so it's indicated in the diagram as *rdf\_about*.

The namespaces used in the model are the following:

Prefix	Namespace URL	Description
rdf	<a href="http://www.w3.org/1999/02/22-rdf-syntax-ns#">http://www.w3.org/1999/02/22-rdf-syntax-ns#</a>	The RDF namespace
rdfs	<a href="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#">http://www.w3.org/TR/1999/PR-rdf-schema-19990303#</a>	The RDFS namespace
n	<a href="http://www.nesstar.org/rdf/">http://www.nesstar.org/rdf/</a>	The NEOOM namespace
m	<a href="http://www.nesstar.org/rdf/Method/">http://www.nesstar.org/rdf/Method/</a>	The NEOOM Method class namespace
p	<a href="http://www.nesstar.org/rdf/Parameter#">http://www.nesstar.org/rdf/Parameter#</a>	The NEOOM Parameter class namespace

Table 2.1: Object Model Namespaces

The definitions of the RDF and RDF Schema concepts are contained in the RDF [LS99] and RDF Schema [GB00] standards and won't be repeated here. It might be useful though to summarize very briefly the basic principles of the RDF model:

- RDF provides a way of describing WWW resources
- Each resource (or object) has a unique URL
- Each object is an instance of one or more classes (types)
- All the objects of a given class have the same properties

The properties are the attributes of an object, they can be either:

- a literal (a String)
- a reference to another object

## 2.2 Type System

RDF doesn't have a rich datatype system. It is the intention of the designers of the language that RDF will eventually use the datatypes defined in the XML Schema specification. How exactly this is going to happen hasn't been formalized yet (it will probably be one of the points clarified by the activity of the RDF Core Group [RDF]). In the meantime, in order to be able to define the range of properties and parameters and the type of the values returned by method invocations, NEOOM provides some basic datatypes:

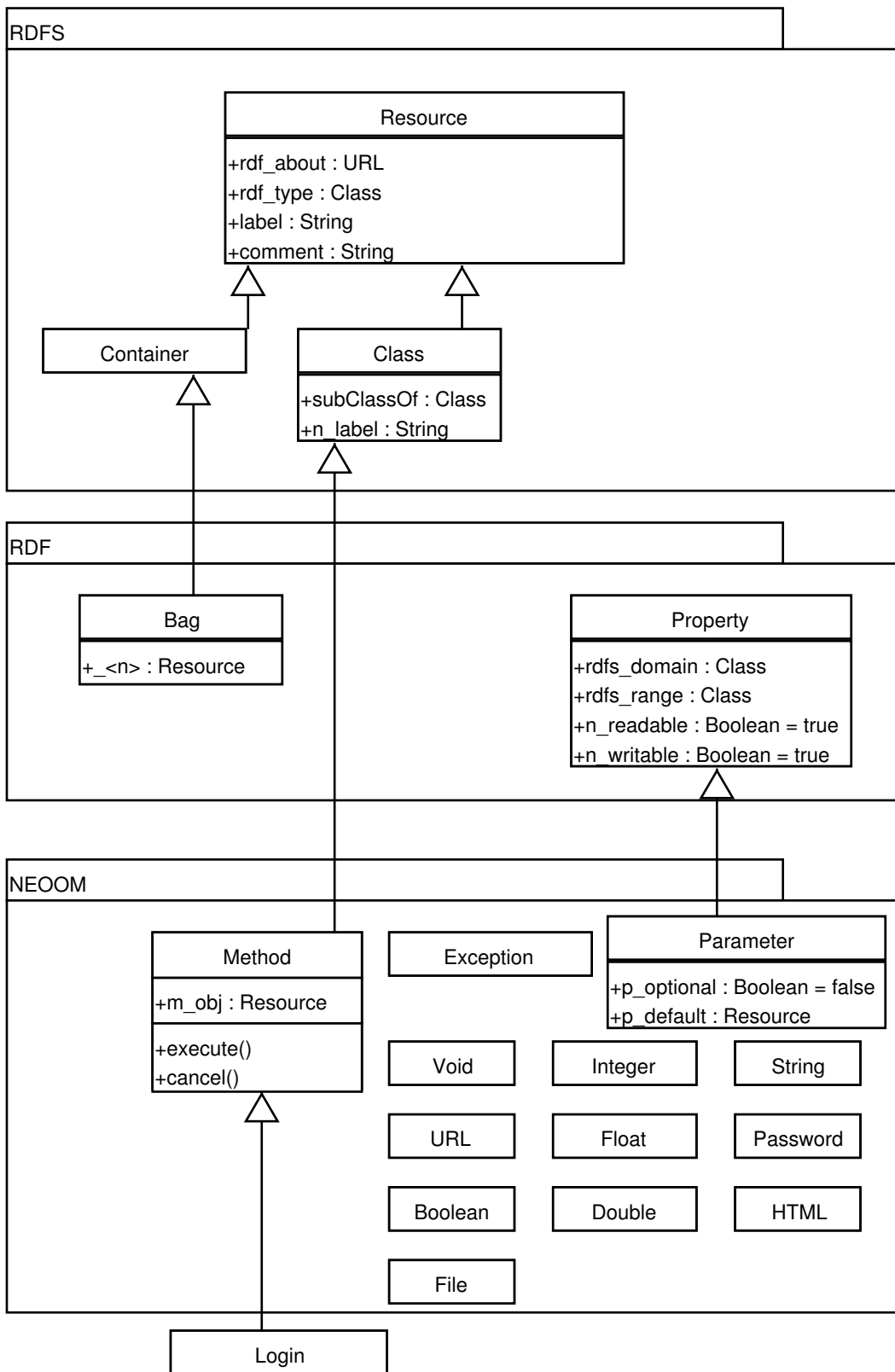


Figure 2.1: NEOOM Object Model

<b>RDF Class</b>	<b>Input/Output</b>	<b>XML Schema equivalent</b>
Void	O	
Boolean	I/O	boolean
URL	I/O	anyURI
<b>Text types</b>		
String	I/O	string
Password	I	string
HTML	I/O	
<b>Numeric types</b>		
Integer	I/O	integer
Float	I/O	float
Double	I/O	double
<b>Binary types</b>		
File	I/O	

Table 2.2: NEOOM primitive types

The table specifies:

- the name of the RDF class that represent the type (all names are in the NEOOM namespace)
- if the type is meant to be used as the range of a method parameter (Input) and/or as the value returned by a method (Output)
- the equivalent XML Schema primitive type, where it exists

In addition to the primitives types NEOOM definitions can make use of the standard RDF container types *rdf:Bag* and *rdf:Seq* [LS99, sect3]<sup>1</sup> and of any other RDF/NEOOM class.

## 2.3 Properties

Object properties are defined in NEOOM exactly as in RDF using the *rdf:Property*<sup>2</sup> class (see sect 2.2.2). NEOOM adds two properties to the *rdf:Property* class<sup>3</sup>:

<sup>1</sup> *Java implementation note:* Only *rdf:Bag* is currently supported

<sup>2</sup> *Java implementation note:* *rdf:Property* is incorrectly assumed to be in the *rdfs* namespace.

<sup>3</sup> *Java implementation note:* A similar property called *n:readOnly*, equivalent to *n:readable=true n:writable=false*, is supported. *n:readable* and *n:writable* are not supported yet.

Property	Definition	Range	Default value
n:readable	Specify if the value of the property can be read.	n:Boolean	true
n:writable	Specify if the value of the property can be modified.	n:Boolean	true

Table 2.3: rdf:Property additional properties

Please note that *n:readable* and *n:writable* express absolute possibilities but they do not specify if the property can be read or written in a particular usage context. This is determined by the access control policy that applies to the object (see section 3.8 and section 4.3).

## 2.4 Methods

An object method is an operation that an object can perform. In NEOOM, methods are defined as subclasses of the *n:Method* class. Method invocations are represented by instances of the method classes.

Defining a method as a class is a bit unusual<sup>4</sup> but it has the advantage of making a method invocation an object in its own right. Being an object a method invocation can be represented in RDF (and therefore stored, transmitted or logged easily) and can have methods and properties. The *n:Method* class has the following methods:

Method	Definition	Range	Parameters
m:execute	Execute the method.	rdfs:Resource	none
m:cancel	Cancel the execution of the method.	n:Void	none

Table 2.4: n:Method methods

*m:execute* and *m:cancel* are used to control the execution of a method. A common scenario where this might be necessary is when objects are subject to some access control mechanism. In this case the caller, before being allowed to execute the operation, might have to satisfy a series of challenges (request for authentication, payment, etc.). The execution of a method would therefore be suspended waiting for the caller to get access, to be refused access or to decide to opt out of the method execution. *m:execute* and *m:cancel* can be used to manage the method invocation in this scenario: to restart a suspended operation after a challenge has been satisfied or to

<sup>4</sup>In Java for example, a method is represented by an *instance* of *nesstar.lang.reflect.Method*, not by a separate class.

cancel it completely (the NEOOM access control protocol is described in 3.8).

The *n:Method* class has the following properties:

Property	Definition	Range	Default value
m:obj	The object to which the method is to be applied.	rdfs:Resource	
n:label	A template string used to create human-readable descriptions of method invocations.	n:String	If absent the value of the <i>rdfs:comment</i> property should be used instead. If <i>rdfs:comment</i> is unspecified then <i>rdfs:label</i> should be used.

Table 2.5: n:Method properties

*n:label* is a string that can contain references to the method parameters of the form:

{parameter\_name}

The human-readable description of a method invocation is obtained by substituting the method parameter references with the actual values of the parameters.

### 2.4.1 Parameters

Method parameters are defined by instances of the *n:Parameter* class. A *n:Parameter* is conceptually very similar to an *rdf:property* and it inherits from it. Given that methods are defined as classes then a parameter is one of the property of its instances. *n:Parameter* has the following properties:

Property	Definition	Range	Default value
rdfs:domain	Indicates the (method) class to which the parameter refers (as specified in [GB00, sect.3.1.4] a parameter can have more then one class as its domain)	rdfs:Class	
rdfs:range	Indicates the class of the parameter value (see [GB00, sect.3.1.3])	rdfs:Class	
p:optional	Indicates if the parameter can be left unspecified in a method call.	n:Boolean	false

Table 2.6: n:Parameter properties

p:default	The default value that will be assumed for the parameter if not explicitly specified in the method call.	rdfs:Resource	
-----------	--	---------------	--

Table 2.6: n:Parameter properties

## 2.4.2 Exceptions

If the execution of a method can either not be initiated, because the caller doesn't have the right to perform it, or cannot be completed for any other cause an exception has to be thrown. The thrown exception must be an instance of the *n:Exception* class.

In most programming languages exceptions are associated with specific methods. This works well with respect to concrete method (as defined in a class). When the method is defined in an interface (or an abstract class) it's often harder to specify an accurate list of exceptions. The difficulty stems from the fact that an interface or abstract method can be implemented by many different concrete classes and each implementation might generate a different set of exceptions. As the NEOOM IDL is meant to define interfaces and not concrete classes the same difficulty applies. Recognizing this fact exceptions are not explicitly associated with a method. Any method is free to generate any exception.

The reason of the exception must be specified in human-readable form in the *rdfs:comment* property of the exception instance.

## Chapter 3

# Network Protocol

### Contents

---

<b>3.1</b>	<b>Requirements</b>	<b>29</b>
<b>3.2</b>	<b>Design Options and Related Work</b>	<b>29</b>
3.2.1	Evaluation	30
3.2.2	Supporting Multiple Network Protocols	31
<b>3.3</b>	<b>Accessing an Object State</b>	<b>31</b>
3.3.1	Accessing a Single Property	31
3.3.2	Accessing the Complete Object State	32
3.3.3	Extended/Reduced Information Transfer	33
3.3.4	Accessing a NEOOM Class Definition	33
3.3.5	Caching	33
<b>3.4</b>	<b>Method Calls</b>	<b>34</b>
<b>3.5</b>	<b>Exceptions</b>	<b>36</b>
<b>3.6</b>	<b>Modifying an Object State</b>	<b>36</b>
<b>3.7</b>	<b>URL Representation of Method Calls</b>	<b>37</b>
<b>3.8</b>	<b>Access Control Protocol</b>	<b>38</b>
3.8.1	Tracking User Identity	39
<b>3.9</b>	<b><math>\pi</math>-Calculus Protocol Specification</b>	<b>39</b>

---

A network protocol is a set of rules that dictates how a remote object can be accessed. The Object Model defined in the previous chapter is abstract and completely protocol-independent. Objects defined according to it might therefore be accessed using a number of different protocols, for example the increasingly popular SOAP [SOA00], plain HTTP or SMTP [Kle01].

### 3.1 Requirements

To satisfy the general requirements of the system stated in 1.6 the network protocol must:

1. provide access to both the remote object state and behaviour as defined by the object class definition
2. be compatible with the current Internet protocols and the most common security restrictions that applies to them, in particular it should be able to tunnel through firewalls that allow only HTTP traffic
3. be accessible from a wide variety of programming languages

Additionally it would be useful if the network protocol could:

1. be compatible with current Web browser technology (in the sense that method calls can be coded and executed using a normal browser)
2. provide a representation of a method call as an URL so that it can be stored, exchanged and reapplied easily

### 3.2 Design Options and Related Work

There are a number of transport protocols that might be used to provide access to NEOOM objects. They fall into three basic categories that, in decreasing order of Web-friendliness and increasing order of complexity, are:

1. the basic HTTP protocol [F<sup>+</sup>99] as augmented by HTML Forms [R<sup>+</sup>99, sect17]
2. XML-based protocols (that use HTTP as their transport protocol)
3. binary protocols such as IIOP [IIO] and Remote Method Invocation (RMI) [RMI]

HTTP+Forms is the basic protocol onto which the large majority of the distributed systems in the Internet are built. Its basic principles are very simple. Entities such as web pages, images, multimedia files, etc. are accessed through HTTP GET method calls at their respective URLs. Users can provide information to dynamic services by filling in forms that are coded and transmitted by the user web browser according to the HTML Form conventions. This protocol is simple but flexible enough to allow for the deployment of sophisticated distributed system. It's well understood and familiar to Internet developers and good libraries to build and parse HTTP and HTML Form calls are available for any significant programming language and operating system.

In the last couple of year a number of XML-based protocols have been proposed (for a summary of the available options see [XMLb]). XML protocols adopt some form of XML schema language as their IDL, encode their remote procedure calls (RPC) as XML elements and use HTTP as their basic transport protocol. The main obstacle to their adoption seems to be the sheer number of available alternatives. The W3C has started working on this problem with an *ad hoc* XML Protocol Group [XMLa]. The standardization process hasn't completed yet but it's already clear that the future standard will be an updated version of the SOAP protocol [SOA01] that already enjoys the backing of major software companies such as Microsoft and IBM. Some support libraries are starting to appear and developer awareness of the technology, though probably rather low at the moment, is certainly going to increase especially given that SOAP is one of the building block of the Microsoft .NET architecture [MNE].

The binary protocols have been developed for traditional OOM system such as CORBA and EJB. They are efficient and sophisticated but they are not as widely accessible (RMI is particular is accessible only from Java code) and they have trouble going across the firewalls deployed by most organizations (HTTP tunnelling is available for binary protocols but is not well standardized and can be expensive and cumbersome to use). Because of these problems binary protocols are normally not used to build open distributed systems. They are mostly seen in distributed systems deployed by a single organization (for example: a bank with a number of branches) or by a closely connected group of organizations (for example: supplier chains).

### 3.2.1 Evaluation

HTML+Forms satisfies all the necessary and additional requirements for the network protocol. SOAP satisfies all the necessary requirements (though the second only partially) but fails to satisfy the additional requirements (SOAP is currently unsupported by most browsers and there is no canonical way of encoding SOAP calls as URLs). Binary protocols fail to satisfy the third necessary requirement (and RMI fails the second as well) and do not satisfy any of the additional requirements.

On the base of these short analysis we can conclude that there are currently two protocols that match the requirements: HTTP+Forms and SOAP. Given that SOAP is more complex, still in flux as a standard, and less supported then basic HTTP we have to decide if its advantages are sufficient to overcome these shortcomings.

SOAP has two main advantages over plain HTTP:

1. it allows for encoding of sophisticates data structures (see [SOA01, sect.5.4])
2. it might provide a degree of interoperability across different OOMs

In most scenarios though there is little need for SOAP's sophisticated data structures. There is an inherent asymmetry in client/server systems: the clients normally make simple calls and receive complex answers. Complex answers can be coded in RDF and simple calls do not require SOAP. Interoperability is a potential major benefit but is still unproven (it's unlikely that the differences between the object models underlying the different OOMs can be overcome simply by adopting SOAP as a common protocol, it's more likely that different SOAP "dialects" will be supported by different OOMs) The conclusion is that, at the current time, HTTP+Forms is the best available option for the default NEOOM network protocol.

### 3.2.2 Supporting Multiple Network Protocols

In some cases it might be useful for an object to be accessible through multiple network protocols. For example if an object is implemented using Java it would be rather simple, and potentially more efficient, to provide RMI access to it. In order to support a new protocol two things are needed:

- an appropriate mapping of the object model to the protocol (similar to the mapping to HTTP+Form described in this chapter)
- a way for an User Agent (UA) to find out the protocols supported by an object

Defining a class for each network protocol can solve the second problem. We might have *RMIObject*, *SOAPObject*, etc. classes. The parameters needed for the connection (ex: port number) would be defined as properties of the network protocol class. By making an object an instance of one or more of these classes we can indicate the protocols it supports. By making all the network classes extend a root *NetworkProtocol* class an UA can find out that an object supports additional protocols even if the UA might be unaware of the specific protocols themselves.

## 3.3 Accessing an Object State

There is a special kind of operation that is so common that warrants special treatment in the protocol and that doesn't need to be explicitly specified in the object interface specification: access to the object state. As we have seen in the Object Model (2) chapter every object has an associated set of properties, the set of the values of these properties is the object state.

### 3.3.1 Accessing a Single Property

In object-oriented programming it's common to provide a separate operation (known as an *accessor* operation) to read the value of each property. For example in Java we might write:

```

class Individual {

    ...

    String _name;

    public String getName() {return _name;}

}

```

Similarly in NEOOM a client can access the value of a single property. In order to do so it's not necessary to define explicitly an accessor operation in the class interface.

The request for an object property value can be performed using a GET HTTP method on the URL:

```
object_url "?" (property_url — property_label)
```

So for example to access the debug property of a Server object, whose *rdfs:label* property is *debug*, located at <http://nesstar.data-archive.ac.uk:80/obj/Server> an UA would use the URL:

```
http://nesstar.data-archive.ac.uk:80/obj/Server?debug
```

The object would return the property value as a MIME entity of the appropriate type (as specified in the Type System section). If the property has type String and it has no value it should be returned as a MIME entity of length 0.<sup>1</sup>

### 3.3.2 Accessing the Complete Object State

The state of an object is given simply by the union of the object property values. It could therefore be accessed by reading all its property one at the time and this is actually the way that object state would normally be accessed in an Object Oriented Language (OOL) such as Java. Unfortunately, given the latency of an HTTP call this procedure would be extremely inefficient (more on this point in the following section). For this reason a way is provided to access the complete object state with a single HTTP request.

The state of a NEOOM object, just as that of an ordinary Web resource, can be retrieved by performing an HTTP GET at the object URL. What should be returned is the RDF description of all the object properties. The object might decide to return only a subset of its properties and/or additional information

---

<sup>1</sup>*Java implementation note:* Access to single properties is not supported. The full state of the object will always be returned even if only one property has been requested.

### 3.3.3 Extended/Reduced Information Transfer

One of the basic tenet of the procedure/method call metaphor, no matter if in a procedural or object-oriented language, is that the callee returns only the information it has been requested: no more and no less. This tenet is to be challenged in a distributed environment. The reason is that while in a local environment the latency of a method call is normally insignificant it can dominate the total execution time of an analogous method call in a distributed environment. For this reason it's extremely convenient to allow the remote object to send, together with the answer to the request, any other information that it might deem useful to the client. Say, for example, that an object has been asked for the value of one of its property and that the value of the property is a list of references to other objects. If the linked objects are locally available and of modest dimension they might be returned together with the property to spare the client the burden of performing additional calls to retrieve them (this case is similar to the very common problem of displaying an HTML page that contains embedded images, this requires multiple requests on the part of the WWW browser as the WWW server is not allowed to send the images together with the page that contains them).

There are also cases when it might be appropriate to return less information than requested. An object might have a particularly bulky property that is only rarely used. If it receives a request for its whole state it might return only that part of its state that can be easily transferred and wait for an explicit request to deliver the bigger or less useful properties.

### 3.3.4 Accessing a NEOOM Class Definition

As NEOOM classes are objects they can also be retrieved by accessing their URLs. As an RDF class has no property that links it to its properties and methods their definitions have to be returned together with the class definition itself.

It's often convenient to keep the full class definition in a single file and to make it available by publishing it through a normal WWW server. In order to make sure that the properties and methods can be retrieved individually as well as together with the class definition they should have URLs of the form:

```
class_url "#" object_label
```

### 3.3.5 Caching

*Web caching is the temporary storage of web objects (such as HTML documents) for later retrieval. There are three significant advantages to web caching: reduced bandwidth consumption (fewer requests and responses that need to go over*

*the network), reduced server load (fewer requests for a server to handle), and reduced latency (since responses for cached requests are available immediately, and closer to the client being served)[Dav].*

Caching the state of NEOOM objects is just as useful as the caching of any other WWW resource. The main problem with caching is how to keep synchronized the cached state with the actual object state. NEOOM objects behave with respect to caching exactly as normal WWW resources. They use the HTTP expiration model [F<sup>+</sup>99, sect.13.2] to specify if and for how long the object state can be cached by a UA.<sup>2</sup>

### 3.4 Method Calls

A method call is performed exactly as the submission of an HTML FORM (as specified in [R<sup>+</sup>99, sect17]) where:

- the ACTION property is equal to the URL of the object that is the target of the method
- there is a (form) parameter of name<sup>3</sup> *http://www.nesstar.org/rdf/method* or *method*<sup>4</sup> whose value is the name of the method to perform
- there is a (form) parameter for at least each non-optional method parameter whose name is the name of the method parameter, whose value is equal to the value of the method parameter and whose form input type is as specified in table 3.1.
- the METHOD property is equal to:
  - GET or POST if no form parameter is of type *file*
  - POST otherwise
- the ENCTYPE attribute is equal to:
  - *application/x-www-form-urlencoded* if no form parameter is of type *file*
  - *multipart/form-data* otherwise

---

<sup>2</sup>*Java implementation note:* Objects cannot specify the allowed caching period. All requests for an object state are returned with the indication that no caching is allowed. The API doesn't make any attempt at refreshing object state hold in the local cache.

<sup>3</sup>By name of an object we mean either its full URL or its *rdfs:label*.

<sup>4</sup>*Java implementation note:* Only the full form of the method parameter is recognized.

The result of a successful method call is a MIME entity whose type depends on the class of the method range as specified in table 3.1.

For example the *Login* method of the *n:Server* class, that has two parameter of type String (username and userPassword) would be encoded as the following HTML form:

```
<FORM ACTION="http://155.245.254.82:4000/obj/Server" METHOD=GET>
<input type=hidden name=method value="Login">
<input type=text size=40 name=userID value="myname">
<input type=password size=40 name=userPassword value="mypwd">
<input type=submit value=execute>
</FORM>
```

In this example the values of the parameters are already set. But naturally they can also be provided by a user. One of the key advantages of the protocol is that any method call can be performed using a standard web browser. The following code shows the structure of a form that implements the *SaveFile* operation, used to upload a file to a *n:Server* object. The path where the file is to be stored and the file to transfer are provided by the user. As this operation has a file as one of its parameters the form METHOD is set to HTTP POST and the *enctype* (encoding type) to *multipart/form-data*:

```
<FORM ACTION="http://155.245.254.82:4000/obj/Server"
METHOD=POST enctype="multipart/form-data">
<input type=hidden name=method value="SaveFile">
<input type=text size=40 name=serverPath>
<input type=file size=40 name=file>
<input type=submit value=execute></FORM>
```

NEOOM/RDF Type <sup>5</sup>	HTML FORM INPUT Type	MIME Output Type
Void		human-readable message in text/html or text/plain
Boolean	text — checkbox <sup>6</sup>	text/plain with values <b>true</b> or <b>false</b>
URL	text	text/plain
<b>Text types</b>		
String	text	text/plain

Table 3.1: FORM INPUT and MIME types

<sup>5</sup>All the NEOOM primitive types are defined in the namespace <http://www.nesstar.org/rdf/>.

<sup>6</sup>*Java implementation note:* Checkboxes cannot be used to code booleans.

Password	password	
HTML	file	text/html
<b>Numeric types</b>		
Integer	text	text/plain
Float	text	text/plain
Double	text	text/plain
<b>Binary types</b>		
File	file	any binary type
Any non primitive RDF class		text/xml or text/rdf

Table 3.1: FORM INPUT and MIME types

### 3.5 Exceptions

Whenever an operation on an object (be it to access the object state or to perform a method call) terminates with an error an exception is generated (see 2.4.2). Exceptions are reported by returning a human readable description of the exception as a MIME document of type *text/plain* or *text/html* with an HTTP status code of class 4XX[F<sup>+</sup>99, sect.10.4] or 5XX[F<sup>+</sup>99, sect.10.5].

### 3.6 Modifying an Object State

To change the value of one or more properties of an object state<sup>7</sup> an UA has to perform a HTTP request equivalent to the submission of an HTML FORM where:

- the ACTION property is equal to the URL of the object whose properties are to be changed
- there is a (form) parameter for each property to modify whose name is the name of the property, whose value is equal to the new value of the property and whose form input type is as specified in table 3.1.
- the METHOD property is equal to:
  - GET or POST if no form parameter is of type *file*
  - POST otherwise
- the ENCTYPE attribute is equal to:

<sup>7</sup>Java implementation note: Only the change of a property at a time is supported

- *application/x-www-form-urlencoded* if no form parameter is of type *file*
- *multipart/form-data* otherwise

If the operation completes successfully the object will return its new state just as after a state access operation (see section 3.3).

So for example to set the property 'debug' of a *n:Server* to the value *off* an UA would perform an operation equivalent to the submission of the form:

```
<FORM ACTION="http://.../obj/Server" METHOD=POST>
<input type=text size=40 name=debug value=off >
<input type=submit value=execute>
</FORM>
```

### 3.7 URL Representation of Method Calls

All the operations on remote objects defined in the network protocol can be represented as URLs. This is a very useful feature as it makes possible to "bookmark" operations and store, transmit and reapply them at a later time (see [Ude] for an interesting discussion of the advantages of URLs as representations of operations).

The URLs corresponding to a method invocation has the following form:

```
object_url "?" "method=" method_name ("&" parameter_name "=" parameter_value)*
```

This format is identical to the one used to code normal CGI calls. The difference is that we are using this format also for methods call that have to be performed using the HTTP POST method. In the case of a parameter of form input type *file* the parameter value is the file name, not the file contents. URLs that contain file parameters need some assistance on the part of the UA or of the object to perform correctly. A NEOOM-aware UA will recognize which parameters corresponds to files, by parsing the method definition, and will perform the operation as specified in section 3.4. If the UA is not NEOOM-aware, as it would be the case of an ordinary Web browser, the execution can be assisted by the remote object. Anytime an object receives a method call where the content of the parameters of type *file* (or any other parameter) is not specified it should return a form corresponding to the requested method call with the value of the supplied parameters already set. The user can then execute the operation by simply submitting the returned form (supplying the missing parameters and/or editing the ones provided).

### 3.8 Access Control Protocol

Though the basic protocol is quite simple things are somewhat complicated by the necessity of providing a flexible access control mechanism. Figure 3.1 shows a typical interaction in an access control scenario. The interaction starts with a UserAgent trying to perform a method call on a remote object. The call is received by a Server that hosts the called object and provides controlled access to it. The Server creates a Method object that represents the method to be performed. It then checks if the UserAgent has the right to execute the method. If this is the case the method is executed and a result (or an error if for any reason the execution doesn't complete correctly) is returned to the UserAgent. If on the contrary the UserAgent is not allowed to perform the method an *access denied* exception is returned.

There is also a third case: the Server does not know if the UserAgent can or cannot perform the operation. Consider for example a simple access control policy that gives only to the user John Smith the right to perform the operation. If the user hasn't yet identified himself the Server is unable to establish if he can execute the operation or not. In this case the Server will return a challenge. A challenge is an operation that, if performed by the user, might give him the right to execute the requested operation. A challenge is represented by a human-readable document, of type *text/html*, containing an HTML form that codes the required operation (login, etc.). In this particular case the Server would send back a login form so that the user can identify himself. Together with the challenge the Server also sends back to the UserAgent the URL of the method object. When the UserAgent detects a challenge (by checking the presence in the header of the Server response of a *x-nesstar-challenge* header, whose value is the URL of the Method object) it will:

- create an instance of a web browser to display the challenge to the User
- perform a second call, whose target is the Method object asking it to execute and return its result

At this point we have two parallel interactions going on. The first is between the User, assisted by the Web Browser, and the Server. The other is between the UserAgent and the Method (with the UserAgent waiting for the final result to be provided).

The Web Browser displays the challenge to the User. The User supplies the required information and clicks on the form submit button. The Web Browser sends back the form to the Server. The form contains a hidden parameter that tells the Server which Method the challenge refers to. The Server performs the operation supplied by the user and checks if the original Method can now be executed. If this is the case the Server asks the Method

to complete the execution. The Method will do so and return the final result, or an error, to the UserAgent. The Server will also return to the Web Browser either a "congratulations" page to notify the User that the operation has been executed successfully or will redirect the Web Browser to the URL specified in the *onAccessGranted* parameter by the UserAgent at the time of the original call. If access cannot be granted the Server will return an access control error to the UserAgent and either an error message to the Web Browser or it will redirect the Web Browser to the URL specified in the *onAccessRefused* parameter, if this parameter was specified in the original call. If access cannot yet be granted it will send back another challenge.

### 3.8.1 Tracking User Identity

In order to perform any kind of identity-based access control it's naturally necessary to be able to track the identity of the user. This can be done using standard HTTP techniques such as cookies [COO] or URL rewriting <sup>8</sup>

## 3.9 $\pi$ -Calculus Protocol Specification

The  $\pi$ -calculus is a very terse notation developed by Robin Milner [Mil99] to model concurrent and mobile systems. The basic principle of  $\pi$ -calculus is very simple: processes communicate through channels exchanging *names* where a *name* is actually nothing but a channel! It might not sound much but Milner has shown that any computational process, and any kind of data, can be expressed using this bare-bones formalism.

The simplicity and expressiveness of this notation make it very attractive for the formal specification of concurrent systems. The NEOOM system is naturally a system of this kind since the idea of modelling its network protocol using the  $\pi$ -calculus or, more precisely, the PICT programming language [PT00]. PICT is based on the  $\pi$ -calculus and adds a number of useful enhancements:

- channels can be used to exchange data structure in addition to simple names, this makes the specification much more concise
- channels are typed and type-checking takes place at compile time, this eliminates many possible errors
- as PICT is a programming language the specification can be executed and tested

---

<sup>8</sup> *Java implementation note:* The server uses cookies to track the user identity.

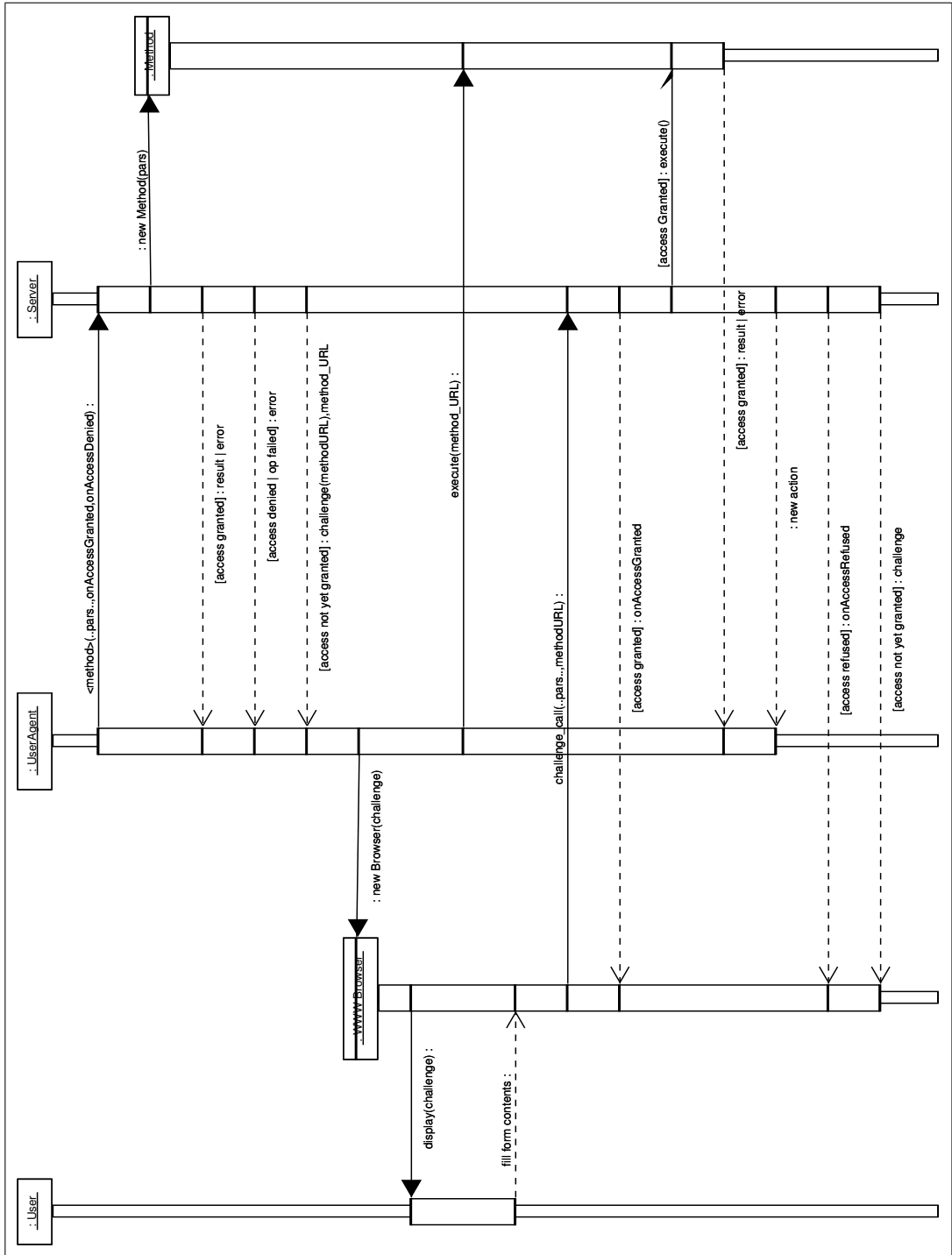


Figure 3.1: Protocol UML Sequence Diagram

The specification is short enough to be included here in full:

```
{- Specification of the HTTP NEOOM network protocol in PICT. -}

type MIMEObject = [mimeType=String content=String challenge=Bool]

type MethodCall = [name=String pars=String onAccessGranted=[] onAccessDenied=[]
  clientIP=String r=~MIMEObject]

def log0 msg:String = ((pr msg); (pr " ["); (pr "]:\n"); ())

def log [call:MethodCall msg:String] = ((pr msg); (pr " ["); (pr call.clientIP); (pr
  "]:\n"); ())

def browser m:MIMEObject = ((pr "Challenge:\n"); (pr m.content); (pr "\n"); ())

def client [obj:^MethodCall ip:String] = (
  new ret:^MIMEObject
  val call=[name="query" pars="survey*" onAccessGranted=[] onAccessDenied=[] clientIP=ip
  r=ret]
  (obj!call
    | ret?r = if r.challenge then browser!r else log![call "Result"])
  )

def object call:^MethodCall = call?c = (log![c "Call"] | op!c | object!call)

and op m:MethodCall =
  (log![m "Return"] | m.r![mimeType="text/html" content="login" challenge=true])

run (
  new 0:^MethodCall

  ( object!0 | client![0 "client1"] | client![0 "client2"] | client![0 "client3"])
  )
```

The specification basically consists in the definition of two typed channels:

**MethodCall** to represent a method call

**MIMEObject** to represent the expected result of the method call, an error or a challenge

And three main processes:

**object** the object to which the call is addressed

**client** the UA performing the method call

**browser** the user's WWW browser, used to display the challenges to the user and to send back the user's replies

The *run* clause contain the initialisation code of the simulation. It starts by declaring a channel that will be used by the clients to get access to the remote object (in more familiar terms this would be the URL of the object). It then creates 4 parallel processes: one object (server) and three clients. The output of the program is the following:

```
Call [client1]:
Return [client1]:
Challenge:
login
Call [client2]:
Return [client2]:
Challenge:
login
Call [client3]:
Return [client3]:
Challenge:
login
```

As it can be seen the specification corresponds only to a subset of the actual protocol but even this initial and modest attempt at representing the protocol in PICT confirms the impression that the  $\pi$ -calculus can be a very useful tool in protocol modelling and analysis.

# Chapter 4

## Java Implementation

### Contents

---

<b>4.1</b>	<b>Introduction</b>	<b>44</b>
<b>4.2</b>	<b>Object Server</b>	<b>44</b>
4.2.1	Requirements	44
4.2.2	Overall Operation	44
<b>4.3</b>	<b>Access Control Unit</b>	<b>45</b>
4.3.1	Requirements	45
4.3.2	Overall Operation	45
4.3.3	Conditions	48
4.3.4	Users	48
4.3.5	Actions	48
<b>4.4</b>	<b>Object Browser</b>	<b>49</b>
4.4.1	Requirements	49
4.4.2	Overall Operation	49
<b>4.5</b>	<b>APIMaker</b>	<b>50</b>
4.5.1	Requirements	50
4.5.2	Overall Operation	50
4.5.3	Format of the Java Client Stubs	50
4.5.4	Format of the Java Server Skeletons	50
4.5.5	Synopsis	54
<b>4.6</b>	<b>Faster Object Server</b>	<b>55</b>
4.6.1	Requirements	55
4.6.2	Design Options	56
4.6.3	Overall Operation	58
4.6.4	Mapping between NEOOM objects and EJB objects	58
4.6.5	NEOOM HTTP Protocol Support	58
4.6.6	ACU Support	58

4.6.7	Development Guide . . . . .	58
<b>4.7</b>	<b>Future Work . . . . .</b>	<b>59</b>
4.7.1	XMI mapping and Argo/UML support . . . . .	59
4.7.2	Secure Socket Layer Support . . . . .	59

---

## 4.1 Introduction

In this chapter we examine the architecture of the Java implementation of the NEOOM system.

The system is composed by four main modules:

**Object Server** hosts and provides controlled access to NEOOM objects

**Access Control Unit** implements and enforces access control policies on NEOOM objects

**Object Browser** universal client

**ApiMaker** client stubs and server skeletons generator

## 4.2 Object Server

### 4.2.1 Requirements

- Host NEOOM objects
- Provide controlled access to the objects as specified by a configurable access control policy

### 4.2.2 Overall Operation

The Object Server acts as an object container/manager. It handles the connections with the clients converting the incoming HTTP requests to object method calls. Before executing an operation verifies that the caller has the right to perform it.

The calls from the clients are received by an instance of the *nesstar.server.ObjServlet* class (see the class diagram in figure 4.1). The *nesstar.server.ObjServlet* verifies that the object that is the target of the operation exists and creates an operation object that represents the method call. All operations objects extend the abstract class *nesstar.server.AOp* class. *nesstar.server.AOp* contains the basic method execution logic to enforce timeouts, suspend and resume the execution of a method, etc.

By default the Object Server contains only the code that implements the object that represents the Object Server itself and that is mainly used to provide administrative methods such as reboot, shutdown, get/save files, etc plus the basic access control objects (see section 4.3). Implementations of other NEOOM classes can be added by packing them in JAR files, adding the JARs to the Object Server classpath and by defining a mapping between the NEOOM class and its Java implementation in the server mapping file. NEOOM objects are created by defining them in RDF in the server configuration file. At startup time the Server reads the configuration file and for each NEOOM object defined creates a corresponding Java object, according to the defined mappings.

The Object Server embeds a complete Tomcat system [TOM]. It can therefore be used as a normal web server to publish HTML pages, images, etc. and to develop Java Servlet and Java Server Pages applications. This functionality can be exploited to create web interfaces for NEOOM objects. An example of this kind of client is the Object Browser (see section 4.4).

The server maintains a log of all the operations executed and of their outcome.

## 4.3 Access Control Unit

The Access Control Unit (ACU) is the module that implements and enforces access control policies on NEOOM objects. .

### 4.3.1 Requirements

The ACU has to:

- check that a user has the right to perform an operation on an object
- guide the user in the process of acquiring the rights needed to perform the operation

### 4.3.2 Overall Operation

Each object has an access condition associated with it. In the Server these conditions are specified in an access control configuration file that is read at startup time. The conditions are themselves objects (that can be protected defining appropriate access conditions).

Before executing a method call on an object the system checks that the condition associated with the object is true by invoking the *nesstar.acu.Condition.isTrue(Op)* method. If the check fails it will call the *nesstar.acu.Condition.nextAction(Op)* method to find out if there is an action that, if performed by the user, might make the condition true.

The fact that the condition is false doesn't necessarily imply that the user cannot be granted access. It might simply be the case that the user hasn't performed yet some action, such as identifying itself, that will allow him to get access. So the role of the ACU is not only to check that the access conditions are verified but also to advise the user on the correct sequence of actions to perform in order to get access.

If, for example, the user hasn't declared his identity yet and the condition is that access can be given only to registered users the proposed action would be a "login". After the user has performed this action the ACU might be able to decide if it can grant access to the required operation. In some cases additional actions on the part of the user (for example: accepting a license agreement) might be required. If the *nesstar.acu.Condition.nextAction(nesstar.acu.Op)* method returns *null* this indicates that the user has definitely no right to execute the required operation. In this case the system will return to the user an *access not granted* error (for more details on the access control protocol see section 3.8).

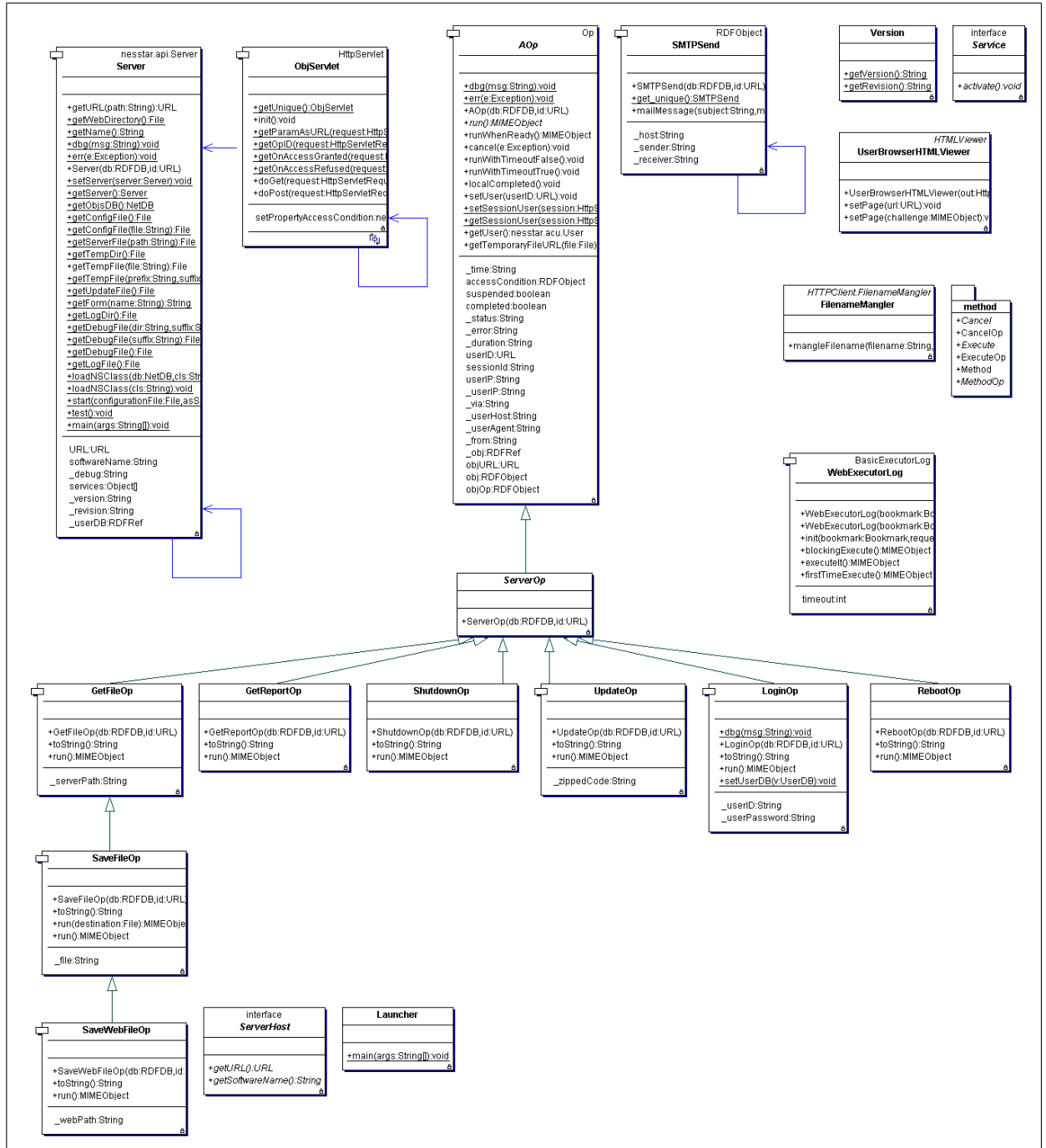


Figure 4.1: Server UML Class Diagram



### 4.3.3 Conditions

All conditions objects extends the *nesstar.acu.Condition* class. Some basic conditions that are implemented in the ACU are:

**nesstar.acu.OpCondition** true if the method being performed is of a given type (for example: Reboot)

**nesstar.acu.rdf.UserClassCondition** true if the user is of a given class (for example: an academic)

**nesstar.acu.rdf.AgreementCondition** true if the user has accepted the terms of a specified agreement

**nesstar.acu.rdf.IPCondition** true if the method call originates from a given IP address

To allow for the construction of complex conditions that combine multiple tests the standard logical operators are also provided:

**nesstar.acu.AndCondition** Logical And

**nesstar.acu.OrCondition** Logical Or

**nesstar.acu.NotCondition** Logical Not

To provide unrestricted access to an object its access condition can be set to the:

**nesstar.acu.TrueCondition** Always true

### 4.3.4 Users

In order to provide controlled access the ACU needs to be able to associate method calls with the identity of the users on behalf of whom the operation is carried out. Users are likely to be characterized in quite different ways in different usage contexts. For this reason the ACU provides only a minimalistic definition of a user. The class *nesstar.acu.User* has only three properties, an *username*, a *password* and a *type*. This basic class can be easily extended to provide site-specific user definitions.

The users objects must be stored in a database object that implements the *nesstar.acu.UserDB* interface. Two standard implementations are provided:

**nesstar.acu.FileUserDB** provide access to user definitions stored in the access control definition file

**nesstar.acu.SQLUserDB** provide access to user definitions stored in a SQL database

### 4.3.5 Actions

The different types of user actions are represented by classes that implements the *nesstar.acu.Action* interface. Some basic actions that are implemented in the ACU are:

**nesstar.acu.LoginOrRegisterAction** asks the user to either identify himself by providing his name and password or to register

**nesstar.acu.rdf.AgreementAction** asks the user to accept the terms of an agreement

The list of actions can be easily extended to payment actions, etc.

The system can convert an action object to a human readable form by calling the *nesstar.acu.Action.getHTML()*. This method returns an HTML page that explains what kind of action is required by the user and, normally, an HTML form for the user to fill in and submit to carry out the action. The system will send this form to the UA so that it can be displayed in the user's web browser.

## 4.4 Object Browser

The Object Browser is a universal client. It can be used to examine any object that conforms to the NEOOM specification. Just like a Web Browser is an indispensable tool to browse the normal web the Object Browser is an indispensable tool to browse the NEOOM object web. The Object Browser can be "pointed" to any NEOOM object and will provide an easy-to-use HTML interface to examine, traverse and modify the value of the object properties and to apply any of the object methods.

### 4.4.1 Requirements

- Provide access to any NEOOM object, its properties and methods
- Work with any reasonably recent (HTML 4.x [R<sup>+</sup>99] compliant) Web Browser

### 4.4.2 Overall Operation

The Object Browser is implemented as a Java Servlet [SER]. It accepts the following parameters:

CGI Parameter	Function
url	the URL of the NEOOM object to browse, if a local reference is given (ex: <i>/obj/server</i> ) it is interpreted as relative to the URL of the server where the Object Browser is hosted
action	if the value is equal to <i>LIST</i> the Object Browser will produce a plain HTML interface, otherwise it will produce a frame interface with the left frame showing a tree of the browsed objects with their properties and methods

Table 4.1: Object Browser CGI Parameters

The Object browsers starts by accessing the object state and identifying the object class. It then reads the object class definition to find out what are the object properties and methods. The RDF object and all the other objects that compose the object class definition (class, property, methods, parameter, etc.) are internally mapped to "viewer" objects: *RDFClassViewer*, *RDFObjectViewer*, *MethodViewer*,

etc. To produce the HTML interface the Object Browser simply ask these objects to *print(..)* themselves on an *HTMLOutput* object (see the UML Class diagram in figure 4.4).

For each writable property the Object Browser will create a form that allows the user to change its value. Similarly for each method it will create the corresponding form so that the user can apply the method. A snapshot of the interface browsing an object of class *n:Server* and showing the form corresponding to the *SaveFile* operation is in figure 4.3.

## 4.5 APIMaker

APIMaker is a generator of Java client stubs and server skeletons for NEOOM objects.

### 4.5.1 Requirements

- Produce, from NEOOM IDL definitions in RDF format, client stubs and server skeletons compatible with the Object Server

### 4.5.2 Overall Operation

APIMaker operates very similarly to the Object Browser. It starts by reading in the NEOOM interface definition and mapping its components (class, property, methods, parameter, etc. definitions) to "coder" objects: *ClassCoder*, *MethodCoder*, *ParameterCoder*, *PropertyCoder*, etc. (see the class diagram in figure 4.5).

To produce the actual stubs and skeletons the APIMaker creates an instance of *JavaClass* to represent the java code that is being generated and asks the *RDFClassCoder* object to generate the java code by calling the *code(JavaClass)* method. The *RDFClassCoder* will in turn asks the *PropertyCoder* and *MethodCoder* objects to code themselves so that all the code is produced.

### 4.5.3 Format of the Java Client Stubs

The client stub corresponding to each NEOOM class is a single class that has accessor functions (set/get) for each of the NEOOM properties and a method for each of the NEOOM methods. The stub methods do not execute directly the corresponding operation on the remote object, instead they return a *Bookmark* corresponding to the operation. The *Bookmark* can be stored internally by the *UserAgent* to be reapplied at a later time. To actually apply the operation the client uses an *Executor*. The *Executor* takes as inputs a *Bookmark* and an *ExecutorLog*. During the execution of the method the *Executor* calls the *ExecutorLog* callback methods (*elapsed(long millisecs)*, *void err(Exception exception)*, *void cancelled()*, *void done(MIMEObject result)*) to signal the events that are taking place. This mechanism provides fine control on the execution of NEOOM methods.

### 4.5.4 Format of the Java Server Skeletons

The server skeletons corresponding to each NEOOM class are composed by:

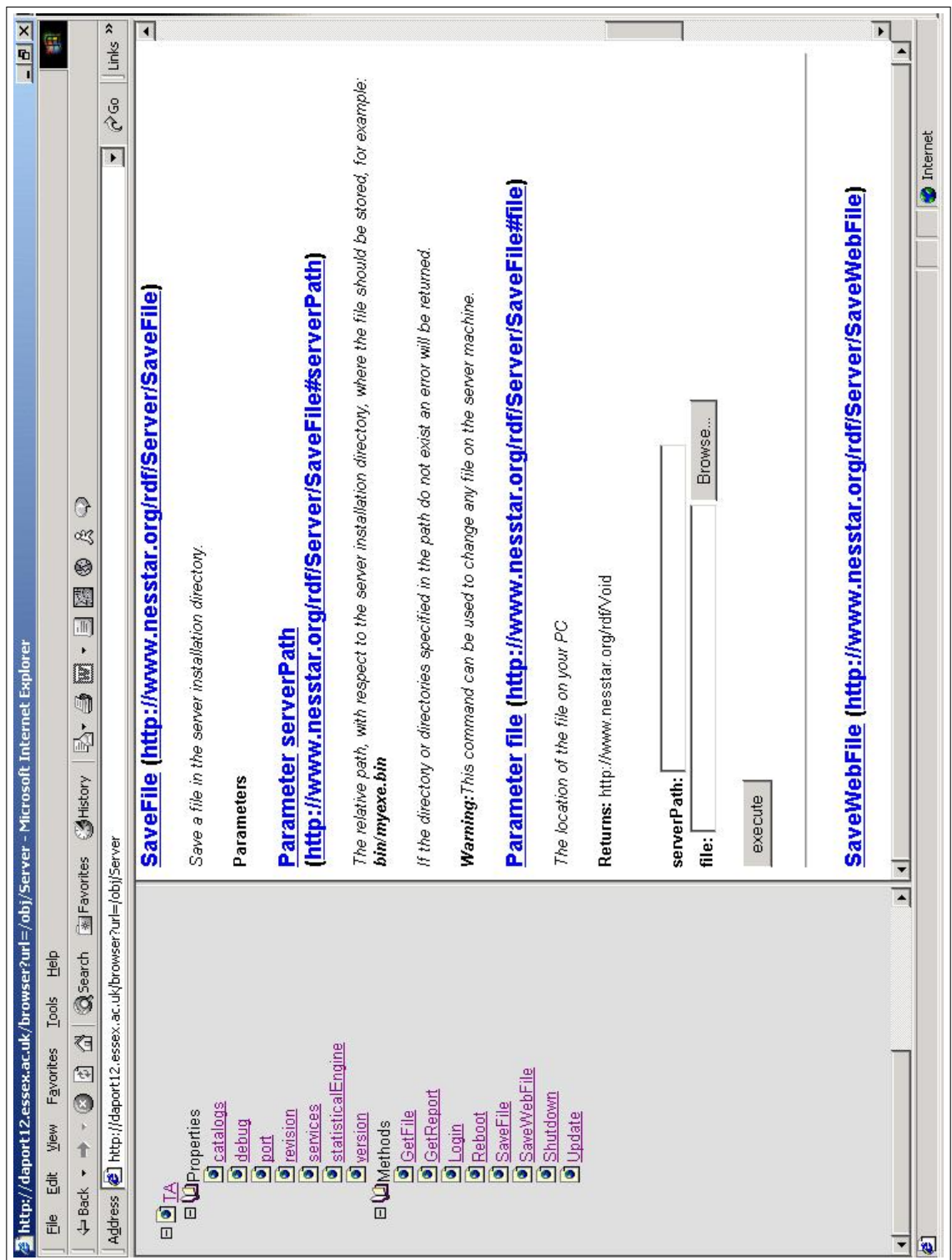


Figure 4.3: Object Browser User Interface

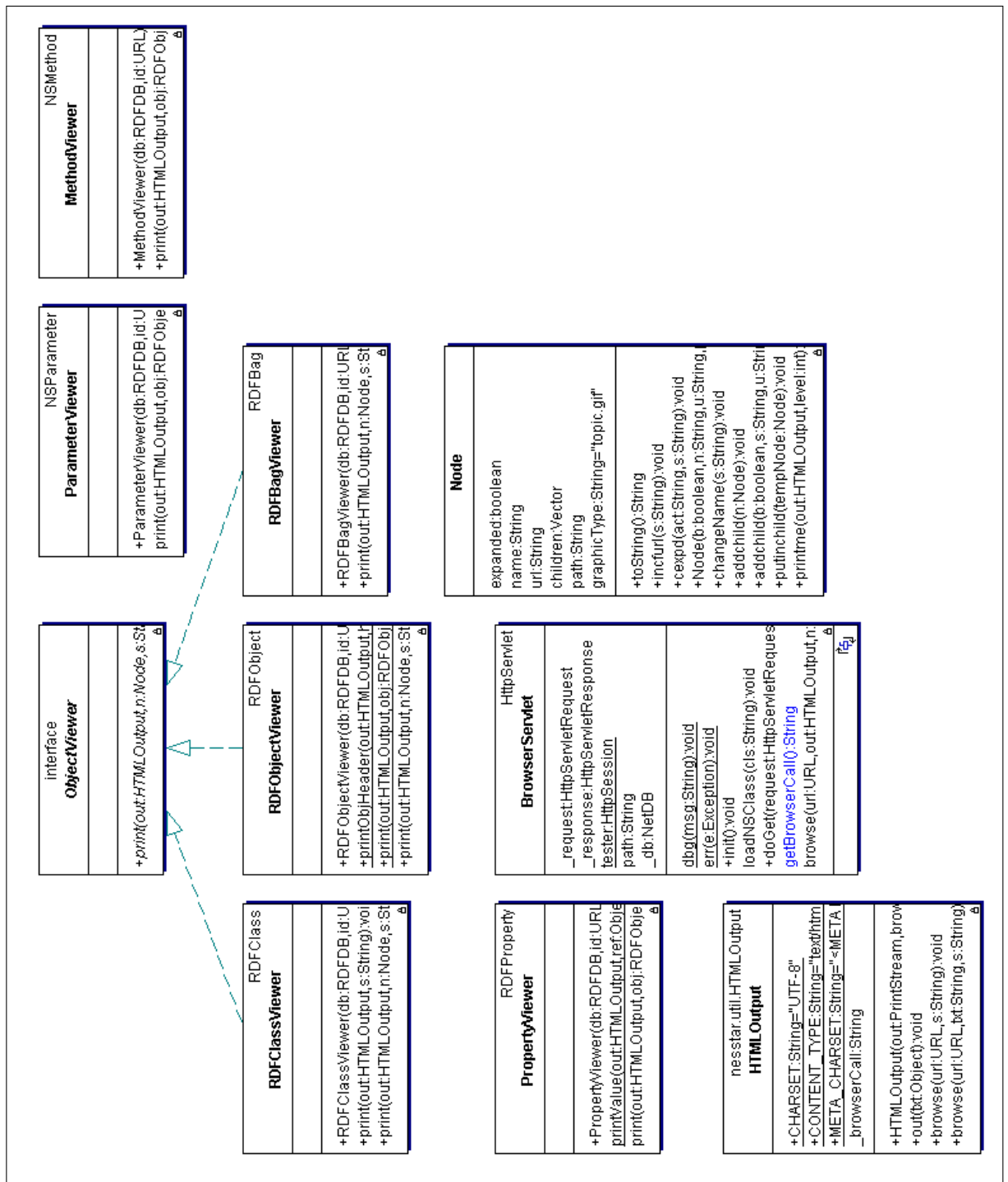


Figure 4.4: Object Browser UML Class Diagram

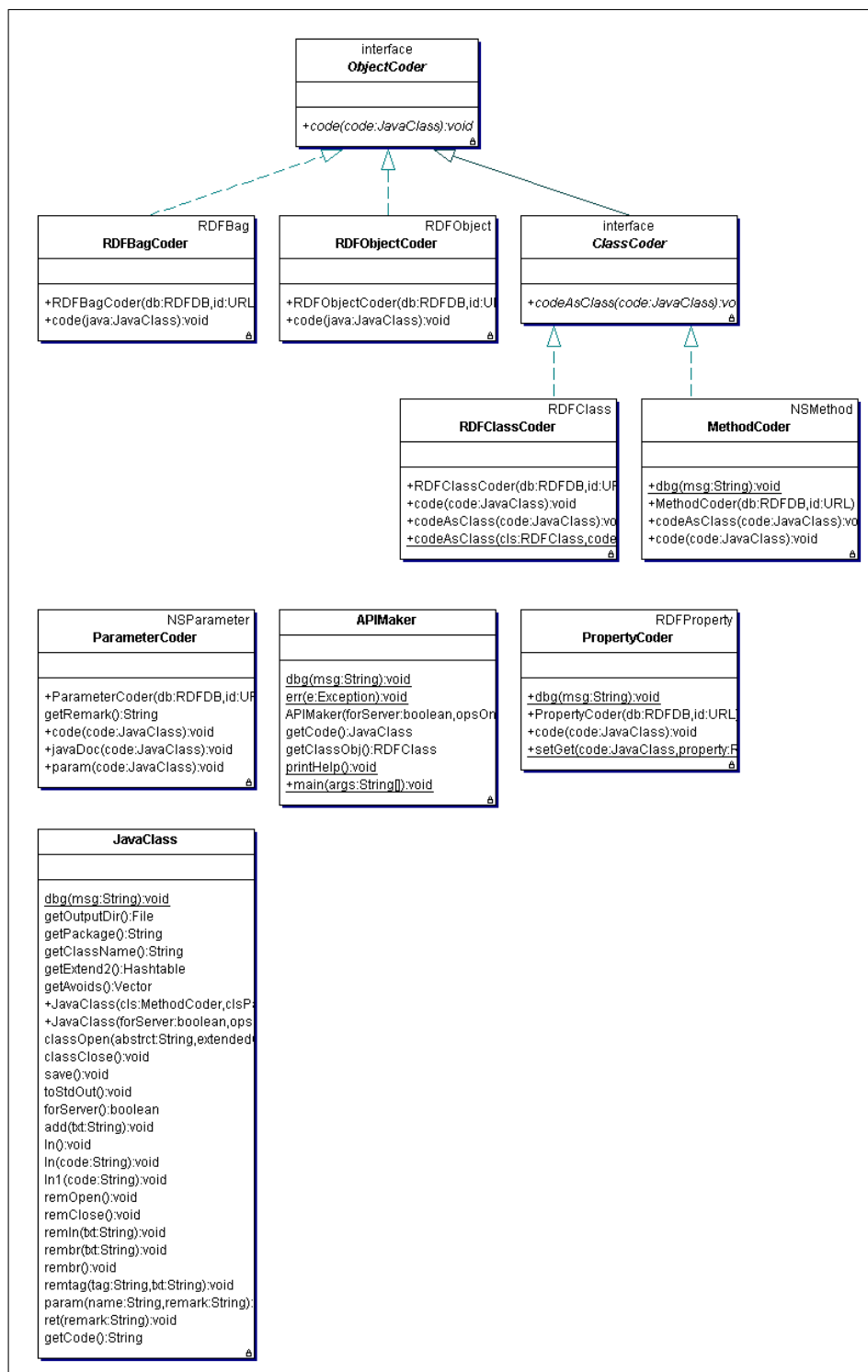


Figure 4.5: ApiMaker UML Class Diagram

- A *state class* that represents the object state, this class has accessor methods for each of the NEOOM class properties
- A *method class* for each of the NEOOM class methods

Examples of both stubs and skeletons (and their implementation) can be found in the source code appendix.

#### 4.5.5 Synopsis

The APIMaker is normally invoked as a command line utility with the following synopsis:

```
java nesstar.apimaker.APIMaker [-server] [-opsOnly]
    -package <package> -rdfClass <rdfClassURL> -outputDir <outputDir>
    [-extend <classExtended>] [-extendMethod <classExtended> <methodLabel>]
    [-avoidMethod <methodLabel>]
```

The APIMaker command line options are explained in table 4.2.

Directive	Description
<b>Common Directives</b>	
-rdfClass <rdfClassURL>	The URL of the RDF/NEOOM class
-package <package>	The package name of the generated classes
-outputDir <outputDir>	The directory where the generated classes have to be created
<b>Directives for the generation of Server Skeletons</b>	
-server	if present generate server skeletons, otherwise generate client stubs
-opsOnly	if present generate only the skeletons for the object methods
-extend <classExtended>	If present the generated state class will extend the <i>classExtended</i> class
-extendMethod <classExtended> <methodLabel>	If present the method class generated for the method whose <i>rdfs:label</i> is equal to <i>methodLabel</i> will extend the <i>classExtended</i> class
-avoidMethod <methodLabel>	If present no method class will be generated for the method whose <i>rdfs:label</i> is equal to <i>methodLabel</i>

Table 4.2: APIMaker Synopsis

## 4.6 Faster Object Server

This section describes a new experimental NEOOM Server being developed for the Faster project. It's still largely incomplete but is being made available to stimulate early feedback in the design phase.

The main reason for rethinking the current server architecture is its lack of proper support for object persistency.

A complete OO application server is composed of three main layers:

1. A network protocol to provide remote access to the objects and to the objects metadata
2. An object container that enforce controlled access (using an ACU) to the objects
3. A persistency layer to store/retrieve object from a datastore

The current server (see section 4.2) provides a reasonable implementation of the first two layers but fails to provide the third layer. Objects can be saved in XML/RDF format and can be stored and retrieved through *ad hoc* JDBC connections but the server itself doesn't facilitate this process.

### 4.6.1 Requirements

Essential requirements:

1. flexible persistency support for NEOOM/Java objects on top of a wide range of relational databases
2. compatibility with the existing NEOOM framework and client software
3. industrial-strength solution (tested, optimised, well-documented, supported)
4. effectively usable in a short time
5. availability of the source code
6. no limitations for commercial usage

Additional (optional) requirements:

1. compatibility with existing standards in order to leverage existing documentation, tools and expertise
2. portability of code across different application server implementations
3. support for non-relational data storages (OO databases, XML databases, etc.)
4. support for popular network protocols (both binary and HTTP/XML based)

## 4.6.2 Design Options

Relevant standards:

**protocol layer** see section 3.2

**container layer** Enterprise Java Beans [EJB]

**persistence layer** EJB and Java Data Objects (JDO) [JDO]

Design options:

1. Adding an object/relational mapping to the existing server
2. Adopting an EJB server with built-in persistence support as the core of the new Server and making it compatible with the NEOOM framework

### Object/Relational Mapping Extension

The first option consists in reusing the current server protocol, replacing the internal object model with a beans-based one and adding a persistence layer using a proven object/relational mapping such as Castor [CAS]. Castor provides configurable support for a number of relational databases (as well as XML files and LDAP servers) and is being made JDO compliant.

### Integration with EJB

The second option consists in using an EJB application server to provide the object container and persistence layer and add to it support for the NEOOM protocol and ACU.

Enterprise Java Beans are the dominant technology for the implementation of multitiered distributed applications in Java. When the development of NEOOM started EJB was considered as a possible choice platform but it was eventually rejected for the following reasons:

- lack of solid open-source implementations
- dependence on a Java specific binary protocol (RMI)
- lack of integration with WWW technology

In the last two years both the specification and the implementations of the EJB architecture have greatly improved:

- Some open-source and freeware implementations are now available
- the dependence on binary protocols has been reduced as it is now possible to connect to EJBs using SOAP or other HTTP-based protocols
- EJB implementations come bundled with a full array of WWW development tools (Servlets, Java Server Pages, etc.)
- EJB server provide support for a wide array of storage backends

We are currently evaluating JBOSS [JBO], a popular open source implementation of the EJB architecture.

JBOSS provides support for two different object/relational mapping: Castor and JAWS.

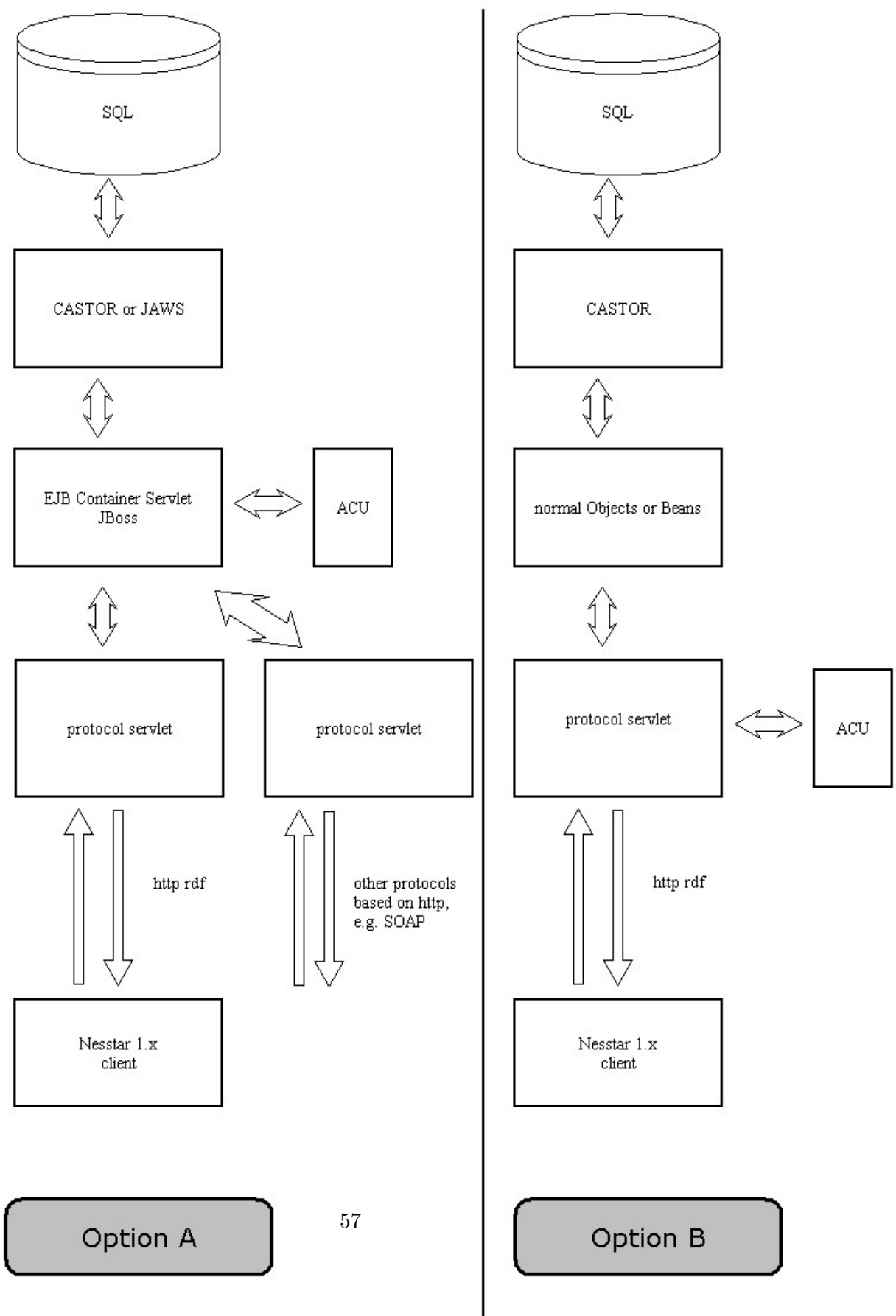


Figure 4.6: Faster Server design options

## Conclusions

...

### 4.6.3 Overall Operation

...

### 4.6.4 Mapping between NEOOM objects and EJB objects

This section details the mapping between NEOOM objects and EJB objects.

...

### 4.6.5 NEOOM HTTP Protocol Support

Compatibility with the NEOOM HTTP Protocol is provided by a Servlet that intercepts HTTP calls, convert them to EJB calls and return the result through HTTP (*neoom.jboss.NEOOM2EJBServlet*).

One problem with EJB is that there seem to be no concept of a unique server-wide object identifier. Entity Beans have unique IDS but these are unique only at the class level. Unique identifiers can nevertheless be produced by combining the entity class Java Naming and Directory Interface (JNDI) [JND] name with the class-specific unique id. This is the approach taken in the current implementation. Objects have URLs of the form:

```
http:// server_url / neoom2ejb_servlet_path JNDI_bean_class_name bean_unique_id
```

For example an instance of ServerBean with JNDI name *Server* might be accessed as:

```
http://daport12.essex.ac.uk:8080/neoomtest/servlet/NEOOM2EJB/obj/Server/1
```

...

### 4.6.6 ACU Support

ACU support is not provided yet. In order to be able to provide secure access to EJB objects regardless of the specific network protocol adopted by the client (NEOOM HTTP, RMI, CORBA, etc) the ACU will have to plug into the existing EJB security architecture: the Java Authentication and Authorization Service (JAAS) [JAA].

One issue to be clarified is if the other network protocols (RMI, CORBA, etc) support, as the NEOOM HTTP protocol, the concept of multiple user challenges.

### 4.6.7 Development Guide

The *ejbserver* directory of the *faster* CVS module contains a first implementation of the JBOSS-based solution. See *guide.html* for details.

## **4.7 Future Work**

### **4.7.1 XMI mapping and Argo/UML support**

Some work is going on to define a mapping between XMI and the NEOOM object model and to integrate an XMI-NEOOM converter in Argo/UML (and possibly its commercial variant Poseidon as well as similar UML case tools). This will allow the usage of Argo/UML to examine existing NEOOM interfaces, generate NEOOM interfaces and Java stubs and skeletons from UML Class Diagrams.

### **4.7.2 Secure Socket Layer Support**

To protect data while in transit on the network is necessary to use some form of cryptography. The simplest option is to add Secure Socket Layer (SSL) support at the protocol level. Tomcat provides SSL support on the server side and there is a standard Java extension to do the same on the client side [JSS].

# Appendix A

## Source Code

### Contents

---

<b>A.1</b>	<b>RDF Interfaces . . . . .</b>	<b>60</b>
<b>A.2</b>	<b>Java Code . . . . .</b>	<b>65</b>
A.2.1	Server . . . . .	65
A.2.2	Access Control Unit . . . . .	67
A.2.3	API Maker . . . . .	70
A.2.4	Object Browser . . . . .	73

---

### A.1 RDF Interfaces

This section lists the RDF definitions that compose the core NEOOM object model.

```
<?xml version="1.0" ?>
<!DOCTYPE rdf:RDF [
<!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
<!ENTITY rdfs "http://www.w3.org/TR/1999/PR-rdf-schema-19990303#" >
<!ENTITY n "http://www.nesstar.org/rdf/" >
<!ENTITY m "http://www.nesstar.org/rdf/Method/" >
<!ENTITY o "http://www.nesstar.org/rdf/Method/obj" >
<!ENTITY usp "http://www.nesstar.org/rdf/User#" >
<!ENTITY parm "http://www.nesstar.org/rdf/Parameter#" >
]>
```

```

<rdf:RDF
  xmlns:rdf="&rdf;"
  xmlns:rdfs="&rdfs;"
  xmlns:usp="&usp;"
  xmlns:n="&n;"
>

<!--                      RDF/RDFS Core Concepts                      -->

<rdfs:Class rdf:about="&rdfs;Class">
<rdfs:label>Class</rdfs:label>
<rdfs:comment>A Class.</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:about="&rdfs;Resource">
<rdfs:label>Resource</rdfs:label>
<rdfs:comment>A generic object.</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:about="&rdfs;Property">
<rdfs:label>Property</rdfs:label>
<rdfs:comment>A property of a Resource.</rdfs:comment>
</rdfs:Class>

<rdfs:Property rdf:about="&rdfs;label">
<rdfs:label>label</rdfs:label>
<rdfs:comment>A single word label.</rdfs:comment>
<rdfs:domain rdf:resource="&rdfs;Resource" />
<rdfs:range rdf:resource="&n;String" />
</rdfs:Property>

<rdfs:Property rdf:about="&rdfs;comment">
<rdfs:label>comment</rdfs:label>
<rdfs:comment>A short presentation of the resource.</rdfs:comment>
<rdfs:domain rdf:resource="&rdfs;Resource" />
<rdfs:range rdf:resource="&n;String" />
</rdfs:Property>

<rdfs:Property rdf:about="&rdfs;subClassOf">
<rdfs:label>subClassOf</rdfs:label>
<rdfs:comment>the Class/Type that this class extends.</rdfs:comment>
<rdfs:domain rdf:resource="&rdfs;Class" />
<rdfs:range rdf:resource="&rdfs;Class" />
</rdfs:Property>

<rdfs:Class rdf:about="&rdf;Bag">
<rdfs:label>Bag</rdfs:label>
<rdfs:comment>A bag: a sequence of objects</rdfs:comment>

```

```

</rdfs:Class>

<!--          NEOOM Core Concepts          -->

<rdfs:Class rdf:about="&n;Method">
<rdfs:label>Method</rdfs:label>
<rdfs:comment>A Method</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:about="&n;Parameter">
<rdfs:label>Parameter</rdfs:label>
<rdfs:comment>A Method Parameter</rdfs:comment>
</rdfs:Class>

<n:Parameter rdf:about="&o;">
<rdfs:label>obj</rdfs:label>
<rdfs:comment>The object to which the method is to be applied.</rdfs:comment>
<rdfs:domain rdf:resource="&n;Method" />
<rdfs:range rdf:resource="&rdfs;Resource" />
</n:Parameter>

<rdfs:Property rdf:about="&n;label">
<rdfs:label>label</rdfs:label>
<rdfs:comment>A template string used to create human-readable descriptions of Method
  invocations.</rdfs:comment>
<rdfs:domain rdf:resource="&rdfs;Class" />
<rdfs:range rdf:resource="&n;String" />
</rdfs:Property>

<!-- Deprecated but still used in the Java implementation -->
<rdfs:Property rdf:about="&n;readOnly">
<rdfs:label>readOnly</rdfs:label>
<rdfs:comment>True if the property is "read only".</rdfs:comment>
<rdfs:domain rdf:resource="&rdfs;Property" />
<rdfs:range rdf:resource="&n;String" />
</rdfs:Property>

<rdfs:Property rdf:about="&n;readable">
<rdfs:label>readable</rdfs:label>
<rdfs:comment>True if the property can be read.</rdfs:comment>
<rdfs:domain rdf:resource="&rdfs;Property" />
<rdfs:range rdf:resource="&n;Boolean" />
</rdfs:Property>

<rdfs:Property rdf:about="&n;writable">
<rdfs:label>writable</rdfs:label>
<rdfs:comment>True if the property can be written (modified).</rdfs:comment>
<rdfs:domain rdf:resource="&rdfs;Property" />

```

```
<rdfs:range rdf:resource="&n;Boolean" />
</rdfs:Property>
```

```
<rdfs:Property rdf:about="&parmp;optional">
<rdfs:label>optional</rdfs:label>
<rdfs:comment>True if the parameter can be left unspecified in a method call
.</rdfs:comment>
<rdfs:domain rdf:resource="&n;Parameter" />
<rdfs:range rdf:resource="&n;Boolean" />
</rdfs:Property>
```

```
<rdfs:Property rdf:about="&parmp;default">
<rdfs:label>default</rdfs:label>
<rdfs:comment>The default value that will be assumed for the parameter if not explicitly
specified in the method call.</rdfs:comment>
<rdfs:domain rdf:resource="&n;Parameter" />
<rdfs:range rdf:resource="&rdfs;Resource" />
</rdfs:Property>
```

```
<!--                                NEOOM Basic Types                                -->
```

```
<rdfs:Class rdf:about="&n;Void">
<rdfs:label>Void</rdfs:label>
<rdfs:comment>A void (absent) result</rdfs:comment>
</rdfs:Class>
```

```
<rdfs:Class rdf:about="&n;File">
<rdfs:label>File</rdfs:label>
<rdfs:comment>A (possibly binary) file</rdfs:comment>
</rdfs:Class>
```

```
<rdfs:Class rdf:about="&n;URL">
<rdfs:label>An URL</rdfs:label>
<rdfs:comment>An URL (for example of a Web page)</rdfs:comment>
</rdfs:Class>
```

```
<rdfs:Class rdf:about="&n;Boolean">
<rdfs:label>Boolean</rdfs:label>
<rdfs:comment>A boolean, a string that can take the values 'true' or
'false'</rdfs:comment>
</rdfs:Class>
```

```
<rdfs:Class rdf:about="&n;String">
<rdfs:label>String</rdfs:label>
```

```
<rdfs:comment>A string</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:about="&n;Password">
<rdfs:label>Password</rdfs:label>
<rdfs:comment>A password</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:about="&n;HTML">
<rdfs:label>HTML text</rdfs:label>
<rdfs:comment>An HTML text</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:about="&n;Integer">
<rdfs:label>Integer</rdfs:label>
<rdfs:comment>An integer</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:about="&n;Float">
<rdfs:label>Float</rdfs:label>
<rdfs:comment>A single precision number</rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:about="&n;Double">
<rdfs:label>Double</rdfs:label>
<rdfs:comment>A double precision number</rdfs:comment>
</rdfs:Class>

</rdf:RDF>
```

## A.2 Java Code

To aid the comprehension of the modules described in chapter 4 this section lists some of their key Java interfaces or classes.

The code has been tested successfully with Java SDK 1.3. It relies, in addition to the standard Java libraries, on a number of external software packages. All of them are freely available in the Internet. They are:

Package Name	Function	Home page
HTTPClient	HTTP Library	<a href="http://www.innovation.ch/java/HTTPClient/">http://www.innovation.ch/java/HTTPClient/</a>
DATAx	RDF Parser	<a href="http://www.megginson.com/DATAx/">http://www.megginson.com/DATAx/</a>
Tomcat	Web Server and Java Servlet Implementation	<a href="http://jakarta.apache.org/tomcat/">http://jakarta.apache.org/tomcat/</a>

Table A.1: Java Libraries

The code has been formatted using the Java pretty-printer *Jacobe* [JAC], according to the Sun Code Conventions for Java [JAV00]. Line breaks have been automatically inserted in lines longer than 90 chars.

The code is the intellectual property of the University of Essex and of Nesstar Ltd, a company funded and owned by the University of Essex and Norwegian Social Science Data Services to exploit commercially the results of the EU-funded NESSTAR and FASTER projects.

### A.2.1 Server

This section includes the following source files:

**nesstar.server.GetFilesOp** An example of a concrete method implementation

**nesstar.server.GetFilesOp**

```
package nesstar.server;

import java.io.*;
import java.net.URL;
import nesstar.api.*;
import nesstar.rdf.*;
import nesstar.util.*;

/**
 * GetFileOp
 */
public class GetFileOp extends ServerOp {

    public GetFileOp(RDFDB db, URL id) throws Exception {
```

```

        super (db, id);
    }

    String _path;
    public String get_serverPath() {
        return _path;
    }

    public void set_serverPath(String s) {
        _path = s;
    }

    public String toString() {
        return "Get server file '" + get_serverPath() + "'";
    }

    public MIMEObject run() throws Exception {
        return new MIMEObject(MIMEObject.APPLICATION_BINARY, Server
.getServerFile(get_serverPath()));
    }
}

```

## A.2.2 Access Control Unit

This section includes the following source files:

**nesstar.acu.Condition** the Condition abstract class

**nesstar.acu.OpCondition** An example condition

**nesstar.acu.Action** The Action interface

### nesstar.acu.Condition

```
package nesstar.acu;

import nesstar.acu.*;
import nesstar.rdf.*;
import nesstar.util.*;

/**
 * An Access Control Condition.
 */
public abstract class Condition extends RDFObject {

    public static void dbg(String msg) {
        DebugLog.dbg(Condition.class, msg);
    }

    /**
     * @param op the operation to apply
     * @return the value of the condition
     */
    public abstract boolean isTrue(Op op) throws Exception;

    /**
     * If the condition is not true returns an action that might make it so, if any.
     * @param op the operation to apply
     * @return an action or null if the condition cannot be satisfied by any additional
     action
     */
    public abstract Action nextAction(Op op) throws Exception;

    /**
     * Simplify the condition tree by removing redundancies
     */
    public void normalize() throws Exception {
    }
}
```

## nesstar.acu.OpCondition

```
package nesstar.acu;

import java.net.URL;
import nesstar.rdf.*;
import nesstar.util.*;

/**
 * The condition that the requested operation is of a give kind.
 */
public class OpCondition extends Condition {

    public OpCondition(RDFDB db, URL id) throws Exception {
        init(db, id);
    }

    URLRef _opClass;
    public void set_opClass(RDFRef url) {
        _opClass = (URLRef) url;
    }

    public RDFRef get_opClass() {
        return _opClass;
    }

    public URL getOpClassURL() {
        return _opClass.getURL();
    }

    public boolean isTrue(Op op) throws Exception {
        boolean result = op.getType().getID().equals(getOpClassURL());

        dbg("OpCondition: opID=" + op.getType().getID() + " classID=" + getOpClassURL() +
" ->" + result);
        return result;
    }

    public Action nextAction(Op op) {
        return null;
    }

    public String toString() {
        try {
            return "the op is " + getOpClassURL();
        }
        catch (Exception e) {
            err(e);
        }
    }
}
```

```

        return null;
    }
}

```

### **nesstar.acu.Action**

```
package nesstar.acu;
```

```
import java.util.*;
import java.io.*;
import java.net.URL;
```

```
import nesstar.rdf.*;
import nesstar.util.*;
import nesstar.server.*;
```

```
/**
 * An action to be proposed to the user with the purpose of making true an access control
 condition, for example: login, accepting some terms, etc.
 */
```

```
public interface Action {
```

```
    /**
     * @return a human readable name for the action, ex: "Login"
     */
    String getName();
```

```
    /**
     * @return an input form to be presented to the user
     */
    String getHTML() throws Exception;
```

```
}
```

### A.2.3 API Maker

This section includes the following source files:

**nesstar.apimaker.ObjectCoder** the main interface implemented by the coders

**nesstar.api.Server** an example of a client stub for a complex class with both properties and methods

**nesstar.server.method.Execute** the server skeleton for the *n:Method* class *execute* method

**nesstar.server.method.ExecuteOp** the implementation of the *execute* method

**nesstar.apimaker.ObjectCoder**

```
package nesstar.apimaker;

/**
 * A converter from NEOOM interface definition objects to Java stub/skeleton code.
 */
public interface ObjectCoder {

    /**
     * @param code the java class being built
     */
    void code(JavaClass code) throws Exception;

}
```

**nesstar.server.method.Execute**

```
package nesstar.server.method;
```

```
import java.net.*;
import java.util.*;
```

```
import nesstar.api.*;
import nesstar.rdf.*;
```

```
/**
 * <br>Execute the method.
```

```

* <br>
* <br>Code automatically generated from APIMaker on Tue Mar 13 14:18:52 GMT+00:00 2001
**/
abstract public class Execute extends MethodOp {

    /**
     * <br>Create an object with the indicated id in the indicated db
     * @param db the db where the object is to be created
     * @param id the id of the object to create
     **/
    public Execute(RDFDB db, URL id) throws Exception {
        super (db, id);
    }

}

```

### nesstar.server.method.ExecuteOp

```

package nesstar.server.method;

import java.net.*;

import nesstar.api.*;
import nesstar.rdf.*;

/**
 * Implementation of Execute
 **/
public class ExecuteOp extends Execute {

    /**
     * <br>Create an object with the indicated id in the indicated db
     * @param db the db where the object is to be created
     * @param id the id of the object to create
     **/
    public ExecuteOp(RDFDB db, URL id) throws Exception {
        super (db, id);
    }

    public MIMEObject run() throws Exception {
        return getOp().runWhenReady();
    }

}

```



## A.2.4 Object Browser

This section includes the following source files:

**nesstar.browser.ObjectViewer** the interface implemented by the viewer objects

### **nesstar.browser.ObjectViewer**

```
package nesstar.browser;

import java.net.*;
import java.io.*;
import java.util.*;

import nesstar.util.*;
import nesstar.rdf.*;
import nesstar.api.*;

/**
 * An HTML viewer for a remote object.
 */
public interface ObjectViewer {

    /**
     * @param out the stream to direct output to
     */
    Node print(HTMLOutput out, Node n, String s) throws Exception;
}

```

# Bibliography

- [And96] Marc Andreessen. *IOP and the distributed objects model*. 1996.
- [Ass00] Pasqualino Assini. Objectifying the Web the 'light' way: an RDF-based framework for the description of Web objects. In *EU Semantic Web Technologies Workshop*, Luxembourg, November 2000.
- [Ass01] Pasqualino Assini. Objectifying the Web the 'light' way: an RDF-based framework for the description of Web objects. In *WWW10 Poster Proceedings*, Hong Kong, May 2001. World Wide Web Consortium.
- [BHL99] Tim Bray, Dave Hollander, and Andrew Layman, editors. *Namespaces in XML*. World Wide Web Consortium, January 1999.
- [CAS] Castor - Open source data binding framework for Java. <http://castor.exolab.org/>.
- [CGI] *The Common Gateway Interface*. National Center for Supercomputing Applications of the University of Illinois.
- [Chr01] Erikothers Christensen, editor. *Web Services Description Language (WSDL) 1.1*. World Wide Web Consortium, March 2001.
- [Col] Mark Colan. *An Overview of Web Services*.
- [COM] Microsoft COM+ Site. <http://www.microsoft.com/com/tech/COMPlus.asp>.
- [COO] *Persistent Client State - HTTP Cookies*. Netscape Inc.
- [COR99] *The Common Object Request Broker: Architecture and Specification; revision 2.3*. The Object Management Group, Framingham (MA), June 1999.
- [DAM01] DAML-S 0.5 - DAML Web Service Ontology. <http://www.daml.org/services/daml-s/2001/05/daml-s.html>, May 2001.
- [Dav] Brian D. Davison. Web Caching and Content Delivery Resources. <http://www.web-caching.com/>.
- [ebX] ebXML Web Site. <http://www.ebxml.org>.
- [EJB] Enterprise JavaBeans. <http://java.sun.com/products/ejb/>.
- [F<sup>+</sup>99] Roy T. Fielding et al. *RFC 2616 - Hypertext Transfer Protocol - HTTP/1.1*. The Internet Engineering Task Force, June 1999.
- [FAS] EU Project FASTER. <http://www.faster-data.org>.

- [GB00] R.V. Guha and Dan Brickley, editors. *Resource Description Framework (RDF) Schema Specification 1.0*. World Wide Web Consortium, March 2000.
- [HLL01] James Hendler, Tim Berners-Lee, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001.
- [IIO] *CORBA/IIOP 2.3.1 specification*. The Object Management Group, Framingham (MA).
- [JAA] Java Authentication and Authorization Service (JAAS). <http://java.sun.com/products/jaas/>.
- [JAC] Jacobe - The Java Code Beautifier. <http://www.tiobe.com/jacobe.htm>.
- [JAV00] *Java Code Conventions*. Sun Microsystems Inc., Palo Alto (CA), 2000.
- [JBO] JBoss - Open source J2EE based application server. <http://www.jboss.org/>.
- [JDO] Java Data Objects (JDO). <http://access1.sun.com/jdo/>.
- [JND] Java Naming and Directory Interface (JNDI). <http://java.sun.com/products/jndi/>.
- [JSS] Java Secure Socket Extension (JSSE). <http://java.sun.com/products/jsse/>.
- [Kle01] J. Klensin, editor. *RFC 2821 - Simple Mail Transfer Protocol*. The Internet Engineering Task Force, April 2001.
- [LS99] Ora Lassila and Ralph R. Swick, editors. *Resource Description Framework (RDF) Model and Syntax Specification*. World Wide Web Consortium, 1999.
- [Man] Frank Manola. Technologies for a Web Object Model. *IEEE Internet Computing*, Special number on Web Object Models(Jan/Feb 1999).
- [Man98] Frank Manola. *Towards a Web Object Model*. February 1998.
- [Mer97] Phillipothers Merrick. *Web Interface Definition Language (WIDL)*. September 1997.
- [Mil99] Robin Milner. *Communicating and Mobile Systems: the  $\pi$ -Calculus*. May 1999.
- [MNE] Microsoft .Net. <http://msdn.microsoft.com/net/>.
- [NESa] EU Project NESSTAR. <http://www.nesstar.org>.
- [NESb] NESSTAR Explorer - Java client for NESSTAR. <http://www.nesstar.org/explorer/>.
- [OHE97] Rober Orfali, Dan Harkey, and Jeri Edwards. *Instant CORBA*. Wiley, New York, 1997.
- [PT00] Benjamin C. Pierce and David N. Turner. Pict: A Programming Language Based on the  $\pi$ -Calculus. In Gordon Plotkin et al., editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Cambridge (MA), May 2000. Massachusetts Institute of Technology.

- [R<sup>+</sup>99] Dave Raggett et al., editors. *HTML 4.01 Specification*. World Wide Web Consortium, December 1999.
- [RDF] W3C RDFCore Working Group. <http://www.w3.org/2001/sw/RDFCore/>.
- [RMI] Java Remote Method Invocation (RMI). <http://java.sun.com/products/jdk/rmi/>.
- [SER] Java Servlet. <http://java.sun.com/products/servlet/>.
- [SOA00] *Simple Object Access Protocol (SOAP) 1.1*. World Wide Web Consortium, 2000.
- [SOA01] *Simple Object Access Protocol (SOAP) 1.2 - W3C Working Draft*. World Wide Web Consortium, July 2001.
- [Tha] Satish Thatte. *XLANG - Web Services for Business Process Design*. Microsoft.
- [TL98] John Tigue and Jon Lavinder. *WebBroker: Distributed Object Communication on the Web*. November 1998.
- [TOM] Tomcat - The reference implementation for the Java Servlet and JavaServer Pages technologies. <http://jakarta.apache.org/tomcat/>.
- [UDD] Universal Description, Discovery and Integration (UDDI) Initiative. <http://www.uddi.org>.
- [Ude] Jon Udell. The Power of the URL-Line. *Byte*, 2001(August).
- [XMLa] XML Protocol Activity. <http://www.w3.org/2000/xp/>.
- [XMLb] *XML Protocol Comparisons*. World Wide Web Consortium.