Project Number	IST-2006-033789
Project Title	Planets
Title of Deliverable	Final version of Service Developers Guidelines
Deliverable Number	IF6-D3
Contributing Sub-project and Work-package	IF/6
Deliverable Dissemination Level	External
Deliverable Nature	Report
Contractual Delivery Date	31st January 2010
Actual Delivery Date	16th March 2010
Author(s)	Carl Wilson

Abstract

This document presents the final version of the Planets Service Wrapping Guidlelines. It is targeted at software developers - within the Planets constortium but also third parties - working in Digital Preservation, who would like to integrate their tools with the Planets software environment.

Keyword list

Interoperability, Web Service, Service Interface, Service Wrapping

Contributors

Person	Role	Partner	Contribution
Carl Wilson	WP Lead	The British Library	Author
Andrew Jackson	WP Member	The British Library	Author

Document Approval

Person	Role	Partner
Christen Hedegaard	SB Member	External Review
Ross King	SP Lead	Internal Review

Distribution

Person	Role	Partner
Public		

Revision History

Issue	Author	Date	Description
1.0	C. Wilson	05.03.10	Initial draft for review
1.1	C. Wilson	15.03.10	Updated for review comments
1.2	C. Wilson	16.03.10	Final version – removed dangling references

References

Ref.	Document	Date	Details and Version

EXECUTIVE SUMMARY

This document presents the final version of the Planets Service Wrapping Guidlelines. It is targeted at software developers - within the Planets constortium but also third parties - working in Digital Preservation, who would like to integrate their tools with the Planets software environment, and make use of applications for Preservation Planning, Testing, and Workflow Orchestration and Execution.

First we introduce the general service categories, for which the various general interfaces are defined. Then we discuss the role of these interfaces in the context of the Planets Data Model and Planets Workflows.

We also clarify the distinction between Planets Services and Workflows. This is an attempt to separate the concerns of tool experts, who should implement simple preservation services based upon wrapped tools, and Planets experts, who have the expertise necessary to integrate tool interfaces and institutional data models through Planets Workflows.

Then we provide some general guidelines for Web Service development, followed by the full interface definitions for the various service categories. There is a working example based upon the development of a Java Fixity characterisation tool. This is taken step by step from start of project to a basic but functional service, adding an element at a time.

We conclude the report with a road map for the future of the Planets code base, service development and information on how developers can become involved in building their own services.

TABLE OF CONTENTS

1. Introduction 6
1.1. Outline 6
2. Planets Service Architecture 7
2.1. Planets IF Service Interfaces 8
2.2. Planets Workflows 8
2.2.1. Institutional Data Models 9
2.2.2. Workflow Documentation 9
3. Technical Guidelines for Tool Wrapping 10
3.1. Introduction to Interoperable Web Service Specifications 10
3.2. How IF Services Leverage Web Service Interoperability 10
3.2.1. Message Optimization Technology 10
3.3. Java Web Services 11
3.4. .NET Web Services 11
4. Planets Preservation Services 12
4.1. Functional Description 12
4.1.1. Characterisation Services 12
4.1.1.1. Fixity 12
4.1.1.2. Characterise 12
4.1.1.3. Format Identification 12
4.1.1.4. Format Validation 13
4.1.2. Preservation Planning Services 13
4.1.3. Preservation Action Services 13
4.1.3.1. Modify 13
4.1.3.2. Emulation 13
4.1.3.3. Format Migration 13
4.1.4. Comparison Services 13
4.1.4.1. Compare 13
4.1.4.2. Compare Properties 13
4.2. Service Datatypes 14
4.2.1. Digital Object 14
4.2.2. Content / Immutable Content 14
4.2.3. Metadata 14
4.2.4. Service Description 14
4.2.5. Service Report 14
4.2.6. Parameter 14
4.2.7. Property 14
4.3. Service Interface Specifications 15
4.3.1. Interface Code Review & Sign Off 15
4.3.2. The PlanetsService Interface 15
4.3.2.1. Interface Name 15
4.3.2.2. Methods 15
4.3.3. Characterisation Services 15
4.3.3.1. Identify 15
4.3.3.2. Validate 16
4.3.3.3. Fixity 16
4.3.3.4. Characterise 16
4.3.4. Preservation Action Services 17
4.3.4.1. Migrate 17
4.3.4.2. CreateView 17
4.3.4.3. Modify 17
4.3.5. Concrete Java Implementation 18
5. Implementing Planets Services 19
5.1. The Service Development Process 19
5.2. Choosing a Tool to Wrap 19
5.2.1. Requirement for the Tool 19
5.2.2. Tool Licensing 19
5.2.3. Ease of Wrapping 19
5.2.3.1. Java API 19

5.2.3.2. Command Line Tools 19
5.2.3.3. Non Java API 20
5.2.3.4. Graphical User Interface Only 20
5.3. Choosing the Interface 20
5.4. Choice of Development Tools and Environment 20
5.4.1. Installing the IF 20
5.4.1.1. From Source 20
5.4.1.2. From the Distribution Package 22
5.4.2. Deploying the IF 22
5.4.2.1. Custom Deployment: A Central Service 22
5.4.3. Installing the Planets Service Project 22
5.5. Creating a Planets Service in Java 24
5.5.1. Unit Tests for pserv Projects 24
5.5.1.1. Unit Test Example 25
6. Creating a Pure Java Fixity Service 27
6.1. Creating the Standard Project Structure 27
6.2. Creating the Ant Build File 27
6.3. The Skeleton Implementation File 29
6.4. Implementing describe() 30
6.4.1. Where to Put the Service Description Information? 30
6.4.2. Creating the Builder 30
6.5. Implementing the calculateChecksum() Method for MD5 Checksums 32
6.5.1. Creating the Objects for Return 32
6.5.2. The JavaDigestUtils Class 32
6.5.3. Digest Algorithm Identification 33
6.5.4. Calculating the MD5 Checksum 33
6.6. Adding Exception Handling 35
6.6.1. NoSuchAlgorithmException 35
6.6.2. IOException 35
6.7. Unit Testing 36
6.7.1. Unit Testing structure 36
6.7.2. Testing describe() 37
6.7.3. Testing MD5 Digest Creation 38
6.8. Adding Support for Multiple Digest Algorithms 39
7. Xena – An Almost Pure Java Service 43
8. A Roadmap for the Service Interfaces 44
8.1. Definition of New Service Interfaces 44
8.2. Definition of New Service Categories 44
8.3. Complex Digital Object Types 44
8.4. Large File Issues 44
8.5. Enhancements for Batching and Asynchronous Service Support 45
8.6. Secure Web Services 45
9. Getting Involved 46
9.1. Software That Might be Wrapped 46

1. Introduction

This document provides a how to guide for wrapping digital preservation software tools to be deployed as part of the Planets Interoperability Framework (IF). A deployed and registered service can be called by The Planets Testbed (TB) or invoked as part of a Planets Interoperability Framework workflow.

The definition and development of preservation service interfaces for the IF has been an incremental process. This has been due to lengthy discussion as to how sophisticated individual services should be, and how much variety in method signatures (method names, return types, and parameters) was acceptable.

Eventually activity concentrated on defining and developing simple, atomic service interfaces that facilitate the job of the service developer.

1.1. Outline

These developer guidelines covers a lot of ground, a rough guide to the document is given below:

- Describe the approach to IF service architecture.
- A Technical Background for IF web services.
- A Description of service interfaces and datatypes.
- A Walkthrough creation of a pserv service, structure, build.xml, unit tests, etc.
- Present a couple of the new interface projects as case studies / examples of how to wrap a tool.
- Outline the quality required of a complete, signed off service (i.e. minimal description, unit tests, etc).
- Discuss best practice for service descriptions (input formats, migration pathways, techreg, parameters, etc.).
- Outline the future direction of IF service interfaces and development.
- Getting involved.

2. Planets Service Architecture

When first implementing a digital preservation service most developers wish to concentrate on low level concepts for simplicity. When implementing format identification services, the action of identifying the format of a single byte sequence is the simplest first step. The developer isn't worried whether the byte sequence is an image from a web page or an image of a page from a book. Although both objects form part of a meaningful whole, the service does not need to be aware of this to perform format identification.

As the workflows a user wishes to design and execute become more sophisticated there is a requirement to express and consider relationships between individual files / byte streams. To support these concepts the workflows must be able to parse and interpret an institutions view of the digital objects it wishes to preserve.

The implications for creating services for the IF were:

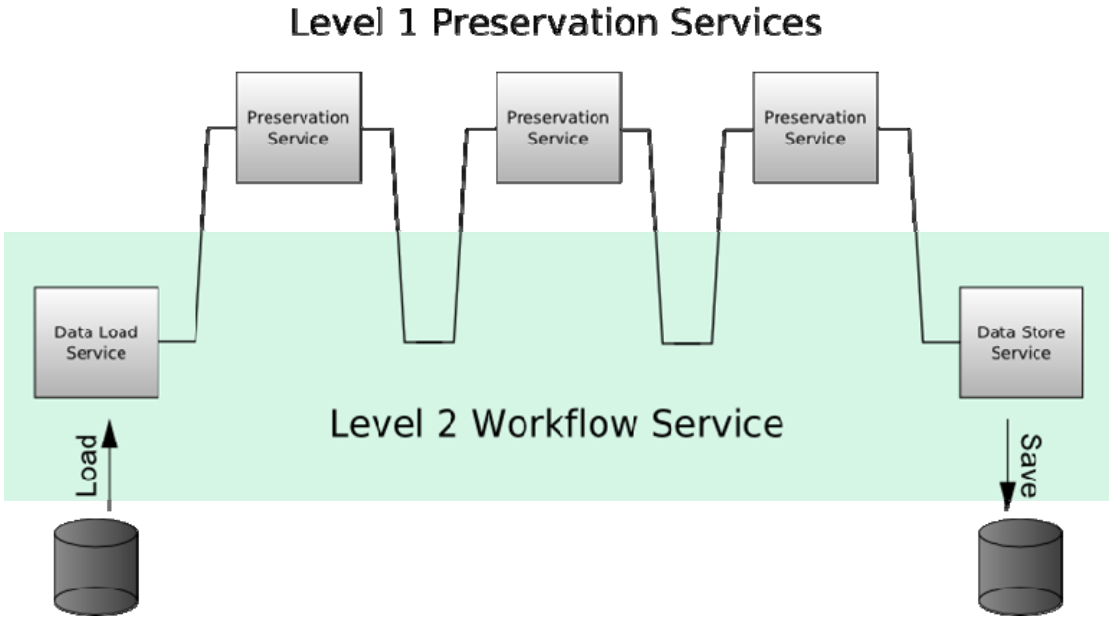
- Different institutions view and model their digital collections in different ways.
- Mapping even simple concepts such as single files to an institutions model can be time consuming and difficult to automate.

To accommodate the tension between a simple, easily implemented interface, and the need to participate in complex, institution dependant workflows a two-tiered service development and implementation was adopted. Developers have the option of creating atomic lightweight services with simple interfaces, **the Planets Interoperability Framework service interfaces**. These allow basic web service wrapping of the tool functionality. They return results and status as simple *struct* types. It is possible to develop and deploy Planets services using a wide variety of environments, programming languages and tools. The interfaces should make sense to developers who understand the function of the tool to be wrapped.

Planets services can, in turn, be wrapped within Planets IF workflows. These workflows:

- Parse institutional data model instances, see Section 2.2.1.
- Parse file references, and retrieve files from institutional data stores.
- Create Planets Digital Objects.
- Call Planets preservation services.
- Transform metadata, and Digital Objects returned from services back to institutional models.
- Persist metadata and data back to institutional data stores.

This diagram shows the relationship between Planets workflows and services:



Note that the load and save operations shown could be to and from a wide variety of sources, likely be some kind of repository software, but it could be as simple as a file system.

2.1. Planets IF Service Interfaces

A basic aim of the Planets Interoperability Framework is to make the wrapping of third party tools so they can be called from Planets IF workflows simple. The aim is to provide a set of light weight interfaces that can be implemented quickly and cut straight to the functionality of the tools to be wrapped.

The Planets development community have created a set of simple, low level interfaces which share some common features:

- Operations are atomic.
- Operations have no requirement to store state.
- Planets Service Datatypes are used for parameters and return values.
- Binary data is handled using the eu.planets_project.services.datatypes.DigitalObject class.

For the majority of tools selecting and implementing an interface should be quick, straightforward, and independent of environment or programming language.

2.2. Planets Workflows

In order to integrate the results of Planets services within complex workflows, accounting for institutional views of digital objects, and digital preservation metadata institutional workflows need to be developed. These workflows parse and interpret sophisticated digital object representations and map the inputs and outputs of Planets services to and from them.

The IF Repository Integration Workpackage (IF/7) will produce workflows that interact with institutional repositories. These workflows will:

- Retrieve data from institutional repositories.
- Map / transform this data to Planets service datatypes.
- Call Planets services to fulfill the workflows intended function.
- Map / transform the outputs of the Planets services to an institutional model.
- Persist whatever data is appropriate, according to institutional requirements.

Workflows are implemented as workflow templates made available for execution via the web service API of the Planets IF Workflow Execution Engine. There is ongoing work designing utility classes on top of the Planets services. These helper classes should automate critical processes such as metadata handling and aiding a developer in assembling institutional workflows.

2.2.1. Institutional Data Models

Planets services operate upon Planets Digital Object instances. This is a pragmatic interface / implementation that requires mapping to and from an institutions own data model. This is achieved by creating a Planets IF Digital Object Manager. This transforms institutional data into Planets Digital Objects that can be processed by Planets workkflows and services.

2.2.2. Workflow Documentation

This document is currently targeted at developers wishing to wrap preservation tools as Planets services. The IF/7 work package will produce real examples of Workflows. This process started in 2009 and will continue over the remainder of the Planets project. Workflows will have their own guide for the end of the Planets project (May 2010).

3. Technical Guidelines for Tool Wrapping

3.1. Introduction to Interoperable Web Service Specifications

Although web services have been around for some time, efforts to establish best practices and specifications for web service interoperability across multiple platforms, operating systems and programming languages are still ongoing. There are a variety of specifications maintained and supported by various standards bodies and entities.

As a collective whole web service specifications can be grouped under the acronym WS-*, however there is no single authority or owning organization for the term. Many specifications do carry WS- as a prefix and they are generally associated with web services.

Sun and Microsoft are working on initiatives to ensure the interoperability of web service technologies. Web Services Interoperability Technology, known as WSIT, is Sun's Java based implementation of web services specifications, based on Java EE 5. It represents an open-source effort to deliver web services interoperability at the lowest level. Windows Communication Foundation (WCF) is Microsoft's unified programming model for building service-oriented applications and is part of the .NET Framework 3.0.

The two companies are testing WSIT and WCF to ensure that WSIT web service clients can access and consume WCF web services and vice versa.

Further information can be found at the WSIT website: https://wsit.dev.java.net/.

3.2. How IF Services Leverage Web Service Interoperability

The PLANETS service development community's efforts to produce interoperable web services utilize WSIT for Java developers and WCF for .NET developers. Interoperability between two of the largest development communities should be relatively painless.

At the moment the technologies are been used to ensure bottom level interoperability, i.e. that services that comply to the appropriate interfaces can be discovered and called by IF workflows, the Planets Testbed, and test clients.

3.2.1. Message Optimization Technology

Planets services often pass binary data by value which can lead to large payloads. When these are encoded into XML for inclusion in SOAP messages even larger payloads are produced. Base 64 encoding normally adds approximately 33% to the size of a binary object.

Message Transmission Optimisation Mechanism (MOTM) employs SOAP attachments that avoid this overhead. Support for MOTM is transparent providing that the service and client infrastructures support it and are properly configured.

Work is also ongoing to investigate, and incorporate where appropriate, other interoperable specifications related to web services, for example:

- **Web Service Security**
  Web services have historically relied on transport based security, for example SSL. WS-Security provides the means to implement interoperable message confidentiality even when messages pass through intermediaries before reaching their endpoint.
  WSIT also implements Web Services Trust which enables web service applications to use SOAP messages to request security tokens that can be used to establish trusted communications between client and web service.
- **Web Service Transactions**
  The Web Services Transactions specifications define mechanisms for transactional interoperability between web services.
  Care has been taken to ensure that Planets services are atomic and stateless. Workflows may need to support transactions and this will be investigated if there is a clear requirement for this.

3.3. Java Web Services

Java developers should consult the WSIT page https://wsit.dev.java.net/ which is the home of Sun's Metro Web Services Stack project. Sun also provide an excellent WSIT Tutorial at http://java.sun.com/webservices/reference/tutorials/wsit/doc/index.html.

The initial set of reference services developed for the IF are all implemented in Java and use Sun's web service stack.

A set of service interfaces for Java developers to work with are provided in the Planets pserv project in the code repository, a snapshot can be found in the Service Interfaces Appendix. As will be seen in the worked example, these should be used to ensure that the services implemented comply with the service interface and are interoperable.

An example deploying a JAX-WS service with:
https://jax-ws.dev.java.net/nonav/2.1.2m1/docs/jaxws-war.html.

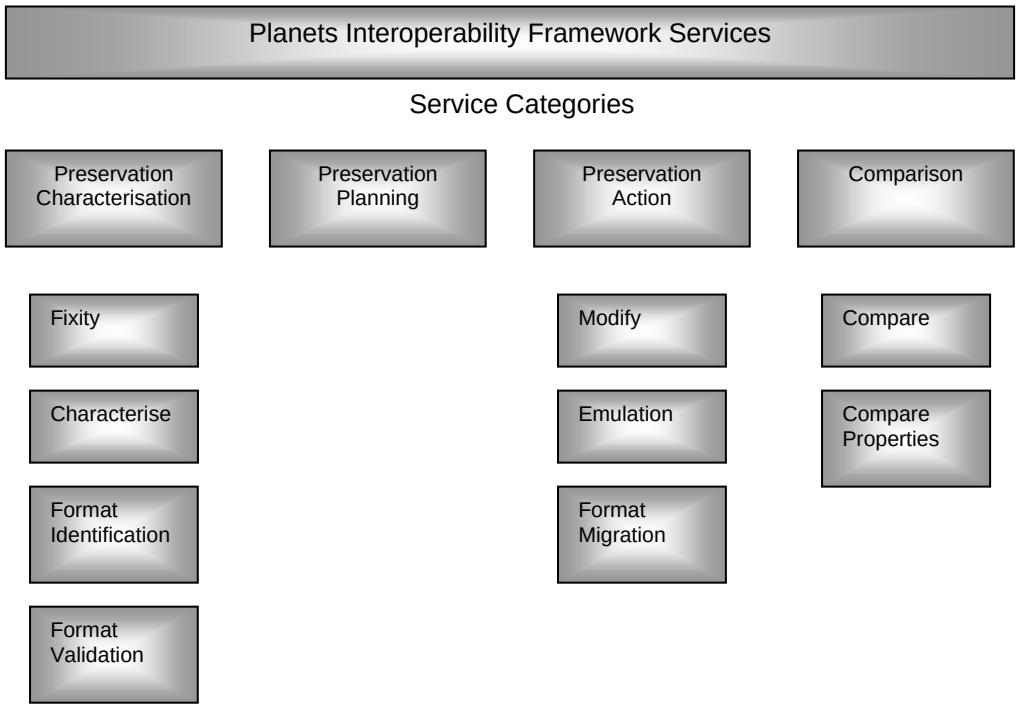
3.4. .NET Web Services

Developers wishing to use Microsoft's .NET development technologies should use the latest version (currently 3.5) of the .NET Framework.Further information is available online at:
http://msdn.microsoft.com/en-gb/netframework/aa663324.aspx.

4. Planets Preservation Services

These are the low level atomic service interfaces that provide a clear, simple framework for wrapping digital preservation tools. Each level one service interface should correspond to a single, indivisible digital preservation verb. The intention is that PLANETS IF service interfaces will provide wrappers for the full range of digital preservation functionality. This means, of course, that the list will never be complete. Advances in digital preservation technology and practices will bring new types of tools and new interfaces.

The Planets Preservation Services can be viewed as a hierarchy, an illustration of which is presented below:



4.1. Functional Description

The following sections provide brief functional descriptions of the level one services envisaged.

4.1.1. Characterisation Services

4.1.1.1. Fixity

Fixity services generate a checksum or digest value that can be used to test whether a digital object has been altered between two points in time. The value generated depends upon the checksum / digest algorithm implemented and can be regarded as a fingerprint for a particular sequence of bytes.

If a fixity value is generated at some point and saved it can then be compared with a recomputed value to ensure that a particular byte sequence has not changed.

4.1.1.2. Characterise

Characterise services measure or extract the properties of a digital object which are significant to its preservation. The significance of particular properties will depend upon the preservation policies of the organization preserving the object.

4.1.1.3. Format Identification

Format Identification services meet a fundamental requirement for any digital preservation infrastructure, to be able to identify the precise format of a byte sequence.

A distinction is drawn between format *identification* and format *validation*. Identification ascertains the format in which an object purports to be encoded whereas validation ensures that the object fully conforms to the format specification.

4.1.1.4. Format Validation

Format validation services determine the level of compliance of a byte sequence to the specification of a particular format. While format identification asks the question “what have I got?” validation asks “is what I have what I think it is?”.

4.1.2. Preservation Planning Services

There is a gap in the project here. It was anticipated that there would be a requirement for Preservation Planning services. One idea was a format risk service that was passed a format URI and returned some numerical risk assessment associated with the format. In practice this information has turned out to be:

- Institutionally specific, i.e. what is considered a risk at for one organisation is not appropriate for another.
- More appropriately added to a format registry, even if that registry had to be specific to a particular institution.

4.1.3. Preservation Action Services

4.1.3.1. Modify

Modify services are used upon files that must be altered but the format isn’t changed. A typical example is a repair service that will normally operate upon a particular file format, or set of formats.

4.1.3.2. Emulation

The Planets Emulation interface is called CreateView. The service does what ever is necessary to provide an environment to render the DigitalObject.

4.1.3.3. Format Migration

Migration services convert data between formats. A digital preservation strategy often involves converting files to current or more accessible formats. This migration might be performed across a whole set of objects as part of a preservation strategy, an organisation might decide to convert all of it's JPEG files to the JPEG 2000 format. The migration can also be performed on demand, when a user requests an image held as a TIFF, it is first converted to a JPEG that is then delivered to the user.

4.1.4. Comparison Services

4.1.4.1. Compare

Compare services compare digital objects and return the difference between them as a list of properties.

4.1.4.2. Compare Properties

Property comparison services compare sets of properties and return the difference between them. The compare properties service makes no judgment of the differences it calculates, it simply returns them. Judgments of differences and application of tolerances are performed by preservation planning services. These atomic steps could be combined by a level two service.

4.2. Service Datatypes

Before looking at the Planets service interfaces an overview of the datatypes that the interfaces operate on is presented. Understanding these types and their use is crucial to implementing Planets services. This section provides links to the Appendices covering the most important of these datatypes.

4.2.1. Digital Object

One of the main problems for the first generation of service interfaces was that all binary data had to be passed by value in a `byte[]`. Support for passing by reference required a `URI` which would have meant creating two interfaces for every service which took a byte sequence as a parameter. This problem is solved by the digital object type which allows developers to pass binary data by reference or value as desired. It also provides support for associating digital preservation metadata with a byte sequence, and recording relationships to other digital objects. See DigitalObject for further details.

4.2.2. Content / Immutable Content

Nearly all digital objects are associated with some kind of binary byte sequence. The Content datatypes are contained within the DigitalObject type and handle the retrieval of the binary data. A content object can handle the byte stream by value, i.e. it holds a copy of the byte stream in a byte array, or by reference, a URL that refers to the byte sequence. It also holds some basic metadata applicable to all byte sequences, length and a Checksum type. See Content for further details.

4.2.3. Metadata

This type is also contained within the DigitalObject class within a List. It is designed to hold the metadata associated with the digital object. It stores the metadata as a String, and also provides for a URI identifier for the metadata type, and a String name that allows identification of multiple metadata blocks of the same type. See Metadata for further details.

4.2.4. Service Description

Used to provide rich metadata about a service, the service description is crucial for service registration, service discovery and providing instructions for installation of required software and using the service. See ServiceDescription for further details.

4.2.5. Service Report

The service report class encapsulates return information common to all types of Planets service. All Planets service return types include a service report in addition to information specific to the service type. See the ServiceReport for further details.

4.2.6. Parameter

Most service interfaces take a set of Parameter objects that allow runtime configuration of the service implementation. The use of parameters is implementation specific. See Parameter for further details.

4.2.7. Property

The Property object is used for multiple purposes. Its primary use is to hold the properties of Digital Objects extracted and measured during characterisation. The object has proved to be multi-purpose and is also used to pass implementation defined information in ServiceReports. This could be performance information, e.g. “wall clock” runtime for the service, or any other data a user may wish to return from their service. The Lists of Property objects in the service Result classes are provided as extensibility points to allow rich information to be returned without altering the interface definition. See Property for further details.

4.3. Service Interface Specifications

This section describes the current set of Java service interfaces. The initial set of deprecated service interfaces are deliberately not documented.

4.3.1. Interface Code Review & Sign Off

There is a desire amongst the IF and service developers to formally quality assure and sign off some of the Java interfaces and data types. The first iteration of this process took place during January 2009. There has been very little re-factoring of the majority of the interfaces and datatypes over the last year. The APIs have remained stable for some time now. There will be a final review of the code and supporting documentation before the end of the project, in May 2010.

4.3.2. The PlanetsService Interface

All Planets service interfaces extend the PlanetsService interface, explicitly every Planets service must implement the PlanetsService interface. The interface methods don't correspond to digital preservation functionality, they specify "meta-methods" required by the Planets IF and service architecture.

4.3.2.1. Interface Name

The fully qualified Java name of the interface is:

```
eu.planets_project.services.PlanetsService
```

4.3.2.2. Methods

The PlanetsService interface currently specifies a single method:

```
ServiceDescription describe()
```

The `describe` method takes no parameters and returns a ServiceDescription. This method is required for automatic service registration, and service discovery / querying by rich service metadata.

4.3.3. Characterisation Services

4.3.3.1. Identify

The Identify interface is to be implemented for Format Identification Services. The interface has been used to wrap several digital preservation tools.

Interface Name

The fully qualified Java name of the Identify interface is:

```
eu.planets_project.services.identify.Identify
```

Methods

The Identify interface only specifies a single method:

```
IdentifyResult identify(DigitalObject digitalObject, List<Parameter>
parameters)
```

The `identify` method takes a DigitalObject, and a List of Parameter objects.

It returns an IdentifyResult which contains a ServiceReport, a List of Format URIs, an identification algorithm Method enum, and a List of Property objects used to return implementation specific data.

The Method enum allows the service to identify the algorithm by which the format URIs were established. Current methods are:

- Extension - the identification was based purely on the file name extension of the digital object.
- Metadata - the identification was based upon other metadata supplied with the digital object, but the bytes sequence wasn't inspected.

- Magic - the identification was based upon matching the digital object's bytes against a bytestream signature.
- Partial Parse - the identification was based upon a detailed inspection of the digital object's bytestream and a significant proportion of the bytes were inspected.
- Full Parse - the identification was based upon a detailed inspection of the digital object's bytestream and every byte was inspected.

Further documentation and the Java interface declaration can be found here.

4.3.3.2. Validate

The validate interface is to be implemented by Format Validation services. Like Identify this interface has been used to wrap multiple tools.

Interface Name

The fully qualified Java name of the Validate interface is:

```
eu.planets_project.services.validate.Validate
```

Methods

The Validate interface only specifies a single method:

```
ValidateResult validate(DigitalObject digitalObject, URI format,
List<Parameter> parameters)
```

The `validate` method takes a DigitalObject, a Format URI and a List of Parameter objects.

It returns a ValidateResult which contains a ServiceReport, a format URI used to indicate the precise format specification that the object was validated against. It also contains a pair of Booleans, one to indicate if the file can be parsed as the given format, a second indicates whether the files is valid with regard to the format specification. Further there are two Lists of Message objects, one that contains a list of all validations errors found by the service, another for validation warnings. Finally there is a List of Property objects used for implementation specific return information, used as an extensibility point.

Further documentation and the Java interface declaration can be found here.

4.3.3.3. Fixity

The fixity interface is to be implemented by Checksum services.

Interface Name

The fully qualified Java name of the Fixity interface is:

```
eu.planets_project.services.fixity.Fixity
```

Methods

The Fixity interface only specifies a single method:

```
FixityResult calculateChecksum(DigitalObject digitalObject,
List<Parameter> parameters)
```

The `calculateChecksum` method takes a DigitalObject, and a List of Parameter objects.

It returns a FixityResult which contains a ServiceReport, and a Checksum Object that holds the checksum value and algorithm identifier. Finally there is a List of Property objects used for implementation specific return information, and as an extensibility point.

Further documentation and the Java interface declaration can be found here.

4.3.3.4. Characterise

The Characterise interfaces is used to wrap tools that extract and measure properties of files and output metadata.

Interface Name

The fully qualified Java name of the Characterise interface is:

```
eu.planets_project.services.characterise.Characterise
```

Methods

The Characterise interface only specifies a single method:

```
CharacteriseResult characterise(DigitalObject digitalObject,
List<Parameter> parameters)
```

The `characterise` method takes a DigitalObject, and a List of Parameter objects.

It returns a CharacteriseResult which contains a ServiceReport, a fragment ID, used to identify a particular part of a multipart object which is typically an object in a zip file. There is also a List of Property Objects, this is for both returning the properties measured, or extracted by the services as well as providing an extensibility point for implementation specific return data. A Format URI is used to indicate the format the service invocation characterised. Finally there is a List of CharacteriseResults, allowing for hierarchical structures of CharacteriseResults for complex objects.

Further documentation and the Java interface declaration can be found here.

4.3.4. Preservation Action Services

4.3.4.1. Migrate

The Migrate interface is to be implemented by format migration services. The current format migration services deal with a single digital object / byte sequence at a time. There is a requirement for format migration interfaces / services that can handle complex digital objects and multiple bytestreams.

Interface Name

The fully qualified Java name of the Migrate interface is:

```
eu.planets_project.services.migrate.Migrate
```

Methods

The Migrate interface specifies a single method:

```
MigrateResult migrate(DigitalObject digitalObject, URI inputFormat, URI
outputFormat, Parameters parameters)
```

The `migrate` method takes a DigitalObject, an initial format URI, a format URI for the result and a list of Parameter objects. The digital object is the object to be migrated, the input and output format URI specify the migration path, the list of parameters allows the service developer to provide control of tool specific functionality.

It returns a MigrateResult object that contains a DigitalObject, which is the migrated version of the object passed to the service. It also contains a ServiceReport.

Further documentation and the Java interface declaration can be found here.

4.3.4.2. CreateView

The CreateView interface was created by the PA/5 workpackage, and is documented in its own deliverable.

Interface Name

The fully qualified Java name of the CreateView interface is:

```
eu.planets_project.services.view.CreateView
```

4.3.4.3. Modify

The modify interface was created for wrapping tools that modify a digital object without changing its format. A typical implementation example would be a service that repaired broken objects of a particular format.

Interface Name

The fully qualified Java name of the Modify interface is:

```
eu.planets_project.services.modify.Modify
```

Methods

The Modify interface specifies a single method:

```
ModifyResult modify(DigitalObject digitalObject, URI inputFormat,
Parameters parameters)
```

The `modify` method takes a DigitalObject, an input format URI, and a list of Parameter objects. The digital object is the object to be modified, the input format URI specifies the format that the object should be treated as, the list of parameters allows the service developer to provide control of tool specific functionality.

It returns a ModifyResult object that contains a DigitalObject, which is the modified version of the object passed to the service. It also contains a ServiceReport.

Further documentation and the Java interface declaration can be found here.

4.3.5. Concrete Java Implementation

The authoritative versions of the IF service interfaces and associated data types can be found in the pserv SVN repository. As the definitions of new interfaces and data types are stabilised the appropriate Java classes and interfaces will be added to SVN.

5. Implementing Planets Services

This section gives practical help for developers wishing to implement a Planets service.

5.1. The Service Development Process

The Planets IF and Service Development teams have established a process for the development, quality assurance and sign off of planets preservation services. This section will concentrate on "section 4.2 Implement" of the process but will say a little about some of the other steps.

The criteria that a new service should satisfy before it can be signed off as complete are listed below:

- Compilation - obviously services that don't compile correctly are unacceptable.
- Interface choice - hopefully this won't be difficult, and the IF Team can help.
- Service Description, the service should provide a minimal acceptable service description, hopefully with some automated tests.
- Quality of service description, it is unlikely that this criterion can be tested automatically, for instance are the provided instructions adequate?
- Unit tests - provision of adequate unit tests that run and pass.

5.2. Choosing a Tool to Wrap

The development process has more to say about this, as does the getting involved section, however the following points are worth emphasizing / repeating.

5.2.1. Requirement for the Tool

Ideally there should be a clear requirement for the tool to be wrapped, whether it's desired by the Testbed, a Planets work package or a colleague at your organisation. Having said that there is sometimes a requirement for a developer to choose a tool and wrap it when testing a new interface specification or as a proof of concept.

5.2.2. Tool Licensing

You should make sure that the intended deployment of the service does not violate the license restrictions of the tool to be wrapped. Open source tools are obviously a good choice here, but even then you should make sure that you consider any other software that the chosen tool depends upon. An open source tool that relies on a Microsoft Word installation would yield a service that is still restricted by the Word licensing terms.

5.2.3. Ease of Wrapping

This is probably most important when developing your first service, or your first service of a particular type. A level of familiarity with the tool to be wrapped will help, but not if, for instance, it only presents a graphical user interface. It may not be possible to accurately asses this criteria up front, but here are some tips we've gathered through experience.

5.2.3.1. Java API

Tools that provide a Java API are a particularly good fit with the chosen IF technologies. Many of wrapped tools in pserv are invoked through a Java API. If the API presented offers the desired functionality and is reasonably well documented then wrapping the tool may be a fairly painless process. This doesn't always help if you are not an experienced Java developer.

5.2.3.2. Command Line Tools

Command line tools can be bespoke wrapped, the Java code to execute a command line statement is straightforward.

The pserv project also provides a Generic Command Line Tool wrapper, which is currently in the final stages of testing. This will receive its own how to guide before the end of the project.

5.2.3.3. Non Java API

This represents trickier territory from Java. The Java Native Interface (JNI) provides facilities will help here but we currently have no examples and little experience in this area. JNI is reputedly not for the faint hearted. Another solution would be to develop the service in another language, for example .NET to call functions in a Windows DLL.

5.2.3.4. Graphical User Interface Only

Currently our best advice here is don't consider it. It may well be impossible, it is certainly very difficult. Choose another tool for the job that presents an API or has a command line interface.

5.3. Choosing the Interface

To some degree the choice of interface is dictated by the functionality desired and the tool chosen. Many tools will be capable of providing multiple services with different interfaces. ImageMagick is an example of a tool that can be wrapped with a Migrate, Characterise, or even a Modify interface as it provides rich functionality.

The Planets services are defined on a class-basis. i.e. each service endpoint implements a standard interface, as defined in the Planets service interface classes. This is done because it makes invoking any Planets service trivially easy - the Planets interfaces can be used as stubs and invocation does not require exploration of the WSDL.

This also means that the operations on each endpoint are already defined, and if you try to add more on a particular service instance, it will not be possible to invoke them. Certainly, extra service metadata is useful, but describing the operation is not necessary as the interface that the endpoint implements defines the operation completely.

For now we will say that the taxonomy is controlled, in that the taxonomy value you assign to a service identifies the interface the service implements, and so you know how to invoke it.

5.4. Choice of Development Tools and Environment

The plain truth is that Java is the best choice of language here, the IF and pserv projects are Java projects after all. We are aware that our support for .NET developers has been somewhere between poor and non-existent. In truth a lot of the effort of dragging two sets of code through the birth pains of the interfaces would have been wasted. We have a .NET service provided by Dialogika that is discoverable and callable by the IF. This really provides a proof of concept for .NET / JAX web service interoperablity.

Currently the IF, and nearly all services are developed as EJBs and deployed on a JBOSS application server. If your choice of tool makes this a feasible deployment model, then we can make your life easier. Downloading the if_sp and pserv projects from SVN would be the best place to start, instructions are given below.

The pserv project also provide Eclipse friendly `.classpath` and `.project` files, making Eclipse a solid choice of IDE.

None of the technologies and tools mentioned above are required to develop a Planets service, but they are tried and tested. To re-emphasise, we are keen to test the deployment of services on a greater variety of technologies / environments.

5.4.1. Installing the IF

If you wish to develop Planets software, we currently recommend that you install the IF from source. The distribution package does not currently address the needs of developers very well. You'll also need an account at the Planets Gforge instance, but the code is been moved to a public SourceForge site in the near future.

5.4.1.1. From Source

Instructions on installing the Interoperability Framework can be found in this INSTALL_IF.txt file. You will need to be able to use Subversion, Java 5, and Apache Ant.

You'll need to check out the if_sp project from the SVN repository:

```
svn co http://gforge.planets-project.eu/svn/if_sp/trunk/ if_sp
```

and pserv:

```
svn co http://gforge.planets-project.eu/svn/pserv/trunk/ pserv
```

In `if_sp` the properties file to create your own configuration:

```
cp framework.properties.template framework.properties
```

Edit `framework.properties` and change `framework.test.dir` to point to the directory where you wish to install the IF, e.g.

```
framework.test.dir=C:/planets_ifr_server/
```

Change the `framework.config.dir` to point to the directory where you want the Planets configuration properties files to live, e.g.

```
framework.config.dir=C:/planets_ifr_server/server/default/conf/planets
```

While still in the `if_sp` checkout dir run:

```
ant deploy-framework
```

Now in pserv, copy the properties file to configure the project:

```
cp build.properties.template build.properties
```

Edit `build.properties` and change the `if_server.dir` property to point at your IF installation, e.g.

```
if_server.dir=C:/planets_ifr_server/
```

This should be the same folder in which you installed the IF, i.e. `framework.test.dir`. Change the `if.server.conf` property to point to the directory where you want the Planets configuration properties files to live, e.g.

```
if_server.conf=${if_server.dir}/server/default/conf/planets
```

This should be the same folder to which you pointed the IF for config files, i.e. `framework.config.dir`.

The user can also set the directory where the Data Registry will look for configuration properties files, e.g.

```
If_server.doms.config.dir=
${if_server.dir}/default/data/planets/dom-config
```

Also change the `if_server.host` and `if_server.port` properties to point at your IF server, e.g.

```
if_server.host=localhost
if_server.port=8080
```

this will allow you to run unit tests against your running IF server.

While in `pserv` run:

```
ant deploy:common
```

Now return to `if_sp`, you may have to run:

```
ant create-ssl-cert
```

if you're not running as localhost, see Custom Deployment below. Finally, again in `if_sp` run:

```
ant create-dbs
```

and you have a valid Planets IF instance.

5.4.1.2. From the Distribution Package

The current binary distribution package is a little out of date in comparison to this document. A final release binary package will be assembled for the end of the project.

5.4.2. Deploying the IF

LOCAL is the default deployment scenario.

Debian:

```
export JAVA_HOME=/usr/lib/jvm/java-1.5.0-sun
export JAVA_OPTS="-Djava.library.path=/usr/local/lib"
```

When running the IF, you may need to add the '-Djboss.platform.mbeanserver'. You do not need to do this if you use the run.sh and run.bat scripts as these include all necessary flags.

For example, in Eclipse under Java 1.6, you will need to use flags like these:

```
-Dprogram.name=run.bat
-Djava.endorsed.dirs="D:/JBOSS_SERVER_Java16/bin/../lib/endorsed"
-Xms256m -Xmx768m
-Djboss.platform.mbeanserver -XX:+CMSClassUnloadingEnabled
-XX:MaxPermSize=256m -Djava.net.preferIPv4Stack=true
-Djavax.net.ssl.trustStore
="D:/JBOSS_SERVER_Java16/server/default/conf/planets.keystore"
```

5.4.2.1. Custom Deployment: A Central Service

To deploy on a central server, the full server name (e.g. 'testbed.hatii.arts.gla.ac.uk') must be placed in the `framework.properties` configuration file. A full re-build of the server is required if this name has been changed.

When running the server, you should not use the -b command to instruct JBoss to bind to a single hostname.

The SSL certificate should also be recreated to reflect the hostname. See the relevant ant task.

The email server should be configured appropriately.

Server context may require `JAVA_OPTS="-Djava.awt.headless=true"` to run without complaining about being unable to start the AWT/Swing.

If you are behind a web proxy, the proxy must be specified on start-up, with explicit exceptions for localhost and 127.0.0.1 so that the server can send local requests (e.g. JOSSO handshake) without going through the proxy. e.g. under a bash shell, you can set your JAVA_OPTS prior to running JBoss. Here is the British Library setup:

```
export JAVA_OPTS="-DproxySet=true -Dhttp.proxyHost=bspcache.bl.uk -
Dhttp.proxyPort=8080 \
   -Dhttp.nonProxyHosts=”localhost|127.0.0.1|*.ad.bl.uk"
```

The nonProxyHosts stuff is very important, as explained above.

HATII central instance setup:

```
set JAVA_OPTS="-DproxySet=true -Dhttp.proxyHost=130.209.6.40 -
Dhttp.proxyPort=8080 \
-Dhttp.nonProxyHosts =localhost|127.0.0.1|testbed.hatii.arts.gla.ac.uk"
```

5.4.3. Installing the Planets Service Project

Once you have installed the Interoperability Framework and verified that it is working, you can deploy more services and start to develop your own.

At this point, you can deploy the current Planets Services as follows.

In the `pserv` directory:

```
ant deploy:all
```

or simply the pure Java services:

```
ant deploy:pure-java
```

Alternatively, you can deploy only those services you require by editing the top-level `build.xml`, or by going into the directory for each set of services and deploying them individually. e.g. for the Xena services:

```
cd PA/xena
ant
```

Given that the application server is running, you should then be able to go to http://{your.server}:{your.port}/jbossws/services and inspect the deployed services. Usually, this will be at http://localhost:8080/jbossws/services, and the links provided will allow you to inspect the WSDL of the services. You will need to log on to the Planets instance, using either the default user account (name: user, password: user), or admin account (name: admin, password: admin).

While this process can deploy the services themselves, it will not ensure that the resources required by those services are in place. For example, the Xena services require Open Office to be installed, and the code requires its own property to be defined so that Xena can find the installation of Open Office. Be aware that each Service may have its own README with its own instructions.

5.5. Creating a Planets Service in Java

In `pserv`, create a directory for your project. e.g.

```
mkdir PA/my_pa_tool
```

for a Preservation Action tool, or:

```
mkdir PC/my_pc_tool
```

for a Preservation Characterisation tool. Having done this, copy the build.xml file from another project (e.g. `PA/xena/build.xml`) and place it in your tool's directory, along with any existing source files you may have.

The `build.xml` file contains definitions that tell the build system how to build your project:

```
<project name="pa-xena" default="deploy:ejb" basedir=".">
  <property name="appName" value="${ant.project.name}"/>
  <property name="appDir" value="."/>

  <property name="src.dir" value="${appDir}/src/java"/>
  <property name="lib.dir" value="${appDir}/lib"/>
  <property name="src.resources" value="${appDir}/src/resources"/>

...other things that should not need changing...
```

You should change the project name to something appropriate, like 'pc-my_pc_tool', and change the src, lib, and resources directories to point to the directories you wish to use for each. For an example of a 'standard' layout, see the Xena codebase. You can now start to develop a service, and to explore this, we consider DROID service as an example.

5.5.1. Unit Tests for pserv Projects

You should have at least two unit tests in mind for your service:

- the `describe` method.
- one test per interface method implemented, most interfaces only specify a single additional method.

The availability / quality of unit tests will be one of the quality assurance criteria for service sign off. We won't be insisting on perfection, but we do want to see the service functionality tested properly.

Within the ant build file build.xml the following section sets up the unit testing:

```
  <property name="test.src.dir" value="${appDir}/test/java" />
  <property name="test.local" value="" />
  <property name="test.standalone" value="" />
  <property name="test.server" value="" />
```

The "test.src.dir" property simply sets the src directory for the unit tests, this is preconfigured and should only be changed by a developer who has special reasons for doing so.

The other three properties set up the testing mode for the build, the values used don't matter, simply the presence of a value triggers the test mode.

```
test.local
```

This defines purely local testing without setting up a web server or web service. This can be used to test the functionality of the code but doesn't test that it works as a web service.

```
test.standalone
```

This testing mode sets up a light, standalone web service that allows unit testing of the service without using the JBOSS application server.

```
test.server
```

This test mode requires a running application server (e.g. JBOSS) at a known URL and port, these ant properties should already be set, for reference they are:

```
pserv.test.host
pserv.test.port
```

Any of the three modes can be invoked from the ant build by choosing one of the following alternative build options:

```
ant test:local
ant test:standalone
ant test:server
```

If you have problems running the tests from Ant, please ensure you have the JUnit task and JUnit4 jar available. We recommend that you use ant 1.7 or later, and that you use ant's '-lib' flag to point to the JUnit4 jar.

5.5.1.1. Unit Test Example

The following shows the unit test configuration set up and used in the DROID Identify service. The pserv projects all follow the same layout convention, the service has a root directory within pserv, in the case of DROID this is pserv/PC/droid. All test code is placed beneath the `test` subdirectory, while any resources required by the project, including tests, are placed in the `src/resources` subdirectory. Some familiarity with Junit is assumed.

The DROID service identifies the format of digital objects passed to it, and the tests focus on this functionality. The DROID application relies on a signature file which is placed in the projects PC/droid/src/resources folder:

```
DROID_SignatureFile_Planets.xml
```

The main DROID tests are located in the DroidTests.java file. The test files are obtained by using the TestFile utility found at:

```
eu.planets_project.services.utils.test.TestFile
```

The first task is to create an instance of the service for testing, this is done using another utility class:

```
eu.planets_project.services.utils.test.ServiceCreator
```

This utility supports the three project test modes, local, standalone, and server without the developer having to worry, the localTests method with the @BeforeClass modifer handles the creation of the service:

```
static Identify droid = null;

@BeforeClass
Public static void localTests() {
       Droid = ServiceCreator.createTestService(Identify.QNAME,
Droid.class, "/pserv-pc-droid/Droid?wsdl");
}
```

A set of single line methods run the individual tests:

```
@Test public void testRtf() { test(TestFile.RTF, droid); }
@Test public void testBmp() { test(TestFile.BMP, droid); }
@Test public void testXml() { test(TestFile.XML, droid); }
@Test public void testZip() { test(TestFile.ZIP, droid); }
@Test public void testPdf() { test(TestFile.PDF, droid); }
@Test public void testGif() { test(TestFile.GIF, droid); }
@Test public void testJpg() { test(TestFile.JPG, droid); }
@Test public void testTif() { test(TestFile.TIF, droid); }
@Test public void testPcx() { test(TestFile.PCX, droid); }
@Test public void testPng() { test(TestFile.PNG, droid); }
@Test public void testWav() { test(TestFile.WAV, droid); }
@Test public void testHtml() { test(TestFile.HTML, droid); }
```

Each of these lines call a single test function:

```
private void test (TestFile f, Identify identify) {
        Assert assertNotNull(“File has no types to compare against: ”
                + f, f.getTypes());
        Boolean b = testIdentification(f, identify);
        Assert.assertTrue(“Identification failed for: ” + f, b);
}
```

6. Creating a Pure Java Fixity Service

The Fixity interface is used to wrap message digest, or one way hash functions. Java provides hash function support from the abstract class:

```
java.security.MessageDigest
```

Full documentation can be found at:

http://java.sun.com/j2se/1.5.0/docs/api/java/security/MessageDigest.html

It's possible to create a basic Fixity service without using any third party tools at all. In this section we'll walkthrough the creation of this service from scratch within the Planets pserv project. The process will be a little long-winded, the aim is to:

- Show how to set up a Planets Service project.
- Implement the `describe()` method, that all services must implement.
- Develop a simple service that calculates MD5 checksums only.
- Allow the user to discover which algorithms are available and choose which to calculate.
- Obtain some basic service metrics and report them to the user.

This will be demonstrated incrementally, finally we'll look at how the service could be further enhanced.

6.1. Creating the Standard Project Structure

The interface is a Preservation Characterisation type, i.e. it provides information about a byte sequence rather than altering. Project convention dictates that characterisation services are placed in a folder under the directory:

```
/pserv/PC
```

The service will be called JavaDigest, so our first step is to create the project directories

```
Service root
/pserv/PC/javadigest

Documentation
/pserv/PC/javadigest/doc

Additional jars required by the service
/pserv/PC/javadigest/lib

Source root
/pserv/PC/javadigest/src

Config, Java packages, and resources for main code
/pserv/PC/javadigest/src/main
/pserv/PC/javadigest/src/main/config
/pserv/PC/javadigest/src/main/java
/pserv/PC/javadigest/src/main/resources

Config, Java packages, and resources for test code
/pserv/PC/javadigest/src/test
/pserv/PC/javadigest/src/test/config
/pserv/PC/javadigest/src/test/java
/pserv/PC/javadigest/src/test/resources
```

6.2. Creating the Ant Build File

Next we need to add an ant build.xml to the project root, create the file:

/pserv/PC/javadigest/build.xml

The basic build file, with extra documentation for clarity, is shown here:

```
<?xml version="1.0"?>
<!-- ====================================== -->
<!-- Planets Services deployer build file    -->
<!-- ====================================== -->

<!— The name attribute is the name of the project preceded -->
<!— by the type of project and a hyphen -->
<!— The default attribute specifies the default build action -->
<!— The basedir attribute should be set to the current dir -->
<project name="pc-javadigest" default="deploy" basedir=".">

        <!-- Specify configuration for this project -->
        <property name="appName" value="${ant.project.name}" />
        <property name="appDir" location="." />
    <!-- Specify the build type: ejb, war or ear. -->
    <property name="pserv.app.build.type" value="ejb"/>

    <!-- Specify if this application requires further software to be
installed -->
    <property name="pserv.app.deploy.mode" value="pure-java" />

        <property name="src.dir" location="${appDir}/src/main/java" />
        <property name="lib.dir" location="${appDir}/lib" />
        <property name="src.resources"
location="${appDir}/src/main/resources" />
        <property name="src.config" location="${appDir}/src/main/config"
/>

    <property name="test.src.dir" location="${appDir}/src/test/java" />
        <property name="test.resources"
location="${appDir}/src/test/resources" />
        <property name="test.config" location="${appDir}/src/test/config"
/>
        <property name="test.local" value="" />
        <property name="test.standalone" value="" />
        <property name="test.server" value="" />

        <!-- Include the build targets etc -->
        <property name="pserv.root.dir" location="../.." />
        <property name="pserv.test.config"
location="${pserv.root.dir}/test/config" />
        <import file="${pserv.root.dir}/build.common.xml" />

</project>
```

6.3. The Skeleton Implementation File

Next we need to create the Java source files for our service implementation. Project convention is that service implementations live at a namespace level below that of the implemented interface. For this project I chose the name JavaDigest, and created the Java source file:

```
/pserv/PC/javadigest/src/main/java/eu/planets_project/ifr/core/services/fixity/javadigest/JavaDigest.java
```

The skeleton implementation file looks as follows:

```
package eu.planets_project.ifr.core.services.fixity.javadigest;

import java.io.Serializable;
import java.util.List;

import javax.ejb.Stateless;
import javax.jws.WebService;
import javax.xml.ws.BindingType;

import com.sun.xml.ws.developer.StreamingAttachment;

import eu.planets_project.services.PlanetsServices;
import eu.planets_project.services.datatypes.DigitalObject;
import eu.planets_project.services.datatypes.Parameter;
import eu.planets_project.services.datatypes.ServiceDescription;
import eu.planets_project.services.fixity.Fixity;
import eu.planets_project.services.fixity.FixityResult;

/**
 * @author Carl Wilson
 */
@Stateless
@WebService(
    name = JavaDigest.NAME,
    serviceName = Fixity.NAME,
    targetNamespace = PlanetsService.NS,
    endpointInterface = "eu.planets_project.services.fixity.Fixity")
@StreamingAttachment(parseEagerly = true)
@BindingType(value =
"http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true")
public final class JavaDigest implements Fixity, Serializable {
    private static final long serialVersionUID =
        -8087686018249395167L;

    /** The name of the service / class */
    static final String NAME = "JavaDigest";

    public FixityResult calculateChecksum(
            DigitalObject digitalObject,
            List<Parameter> parameters) {
        return null;
    }

    public ServiceDescription describe() {
        return null;
    }
```

6.4. Implementing describe()

The first method to implement is the describe() method. All Planets services must implement this method, that returns a ServiceDescription object. This describes the implemented service, and is used by the Service Registry to provide the end user with details of the service.

The Planets service description class is found at:

```
eu.planets_project.services.datatypes.ServiceDescription
```

Like many of the supporting data types, the class is not directly instantiable, it must be created through the supporting builder class:

```
eu.planets_project.services.datatypes.ServiceDescription.Builder
```

An instance of this class can be used to create a ServiceDescription, and a call to its build() method yields an immutable ServiceDescription object.

6.4.1. Where to Put the Service Description Information?

There's usually a lot of static text in the service description. Many services have description information hard coded into either the builder call, which is not the best practise, or as a collection of static final Strings at the top of the service class, which is better.

For this example I decided to implement a utility class to handle the creation of the service description. This class is called:

```
eu.planets_project.ifr.core.services.fixity.javadigest.utils.JavaDigestDe
scription
```

This class then contains a set of static Strings used to create the ServiceDescription:

```
private static final String SERVICE_DESC = "Fixity service based on " +
            Java " + MessageDigest.class.getName() + "\n";

private static final String SERVICE_AUTHOR = "Carl Wilson";
private static final String SERVICE_VERSION = "0.1";
private static final String SERVICE_PROVIDER = "The Planets Consortium.";
private static final String TOOL_DESC = "This MessageDigest class "+
        "provides applications the functionality of the MD5 message " +
        "digest algorithm. Message digests are secure one-way hash " +
        "functions that take arbitrary-sized data and output a " +
        "fixed-length hash value.";

private static final URI SUPPORT_DOCUMENT_LOC =
    URI.create("http://java.sun.com/j2se/1.5.0/" +
               "docs/api/java/security/MessageDigest.html");
```

This does mean that changing any of the description details means recompiling the service but the approach is good enough.

6.4.2. Creating the Builder

The JavaDigestDescription class has one public method:

```
public static final ServiceDescription getDescription()
```

This method creates and returns the populated ServiceDescription for the JavaDigest service. The ServiceDescription is created using the utility Builder class as follows.

First we call constructor for the Builder, we use this to initialise the name and type of the description:

```
// Create a ServiceDescription builder
ServiceDescription.Builder sd = new ServiceDescription.Builder(
        JavaDigest.NAME,
        Fixity.class.getCanonicalName());
```

Next we add the classname property:

```
sd.classname(this.getClass().getCanonicalName());
```

then the description:

```
sd.description(JavaDigest.SERVICE_DESC);
```

For clarity the String passed to the method is defined as a static constant in the service class:

```
private static final String SERVICE_DESC = “Fixity service based ” +
    “on Java “ + MessageDigest.class.getName()”;
```

where MessageDigest is

```
java.security.MessageDigest
```

This describes the underlying digest implementation we’re going to use.

Next we add an author:

```
sd.author(JavaDigest.SERVICE_AUTHOR);
```

Next we add the tool details, this provides details of the software tool / alogorithm used to implement the service:

```
sd.tool(Tool.create(null,
                    MessageDigest.class.getName(),
                    String.valueOf(Java.getVersion()),
                    JavaDigest.TOOL_DESC,
                    JavaDigest.SUPPORT_DOCUMENT_LOC.toString());
```

The tool is created using the:

```
eu.planets_project.services.datatypes.Tool
```

factory method create. The null value is in place of an identifying URI for the tool, this would normally be a value from some kind of tool registry. The other values are the tool name, version, description, and a location for the supporting documentation.

Next we add some further information, a service provider, and a service version number:

```
sd.furtherInfo(JavaDigest.SUPPORT_DOCUMENT_LOC);
sd.serviceProvider(JavaDigest.SERVICE_PROVIDER);
sd.version(JavaDigest.SERVICE_VERSION);
```

Next we add an identifying URI for the input formats that the service can take. The digest service is not sensitive to input format, it only requires a byte sequence. For this we can use the Planets any format URI, which can be obtained from the Format Registy utility classes:

```
import eu.planets_project.ifr.core.techreg.formats.FormatRegistryFactory;

sd.inputFormats(FormatRegistryFactory.getFormatRegistry().createAnyFormat
Uri());
```

Finally the ServiceDescription is built and returned:

```
// Return the service descripiton
return sd.build();
```

The implementation of the describe() method in the actual service implementation file is now as follows:

```
/**
 * @see eu.planets_project.services.PlanetsService#describe()
 */
public ServiceDescription describe() {
    // Call the method from the utility class
    return JavaDigestDescription.getDescription();
}
```

6.5. Implementing the calculateChecksum() Method for MD5 Checksums

The next task is to implement the method to calculate the checksum itself. Initially the service will only support a single digest algorithm, the MD5 checksum algorithm. The signature for the method is the same as that in the Fixity interface:

```
/**
 * @see
eu.planets_project.services.fixity.Fixity#calculateChecksum(DigitalObject
, List)
 */
public FixityResult calculateChecksum(DigitalObject digitalObject,
                                      List<Parameter> parameters)
```

I won't be adding any parameters to the call at the moment as there'll be no user choice.

6.5.1. Creating the Objects for Return

The first step is to create the objects for return, these are a FixityResult and a contained ServiceReport:

```
// The returned FixityResult & ServiceReport
FixityResult retResult = null;
ServiceReport retReport = null;
```

The service will be implemented using Javas built in MessageDigest implementation, this is found in:

```
java.security.MessageDigest
```

6.5.2. The JavaDigestUtils Class

The first step in the code is to create an instance of the MessageDigest class. For clarity I decided to handle some of the implementation details in a utility class:

```
eu.planets_project.ifr.core.services.fixity.javadigest.utils.JavaDigestUt
ils
```

Initially this class deals with some hard-coded details used to select the digest algorithm and create digest algorithm identifiers as URIs, but more on that later. For reference this is the initial implementation of the utility class:

```
/**
 * @author <a href="mailto:carl.wilson@bl.uk">Carl Wilson</a>
 */
public final class JavaDigestUtils {
  /** Util classes providing static methods not to be instantiated. */
  private JavaDigestUtils() {/* Enforce non-instantiability */}
  private static final String ALG_URI_PREFIX = "planets:digest/alg/";
  private static final String MD5 = "MD5";

  /**
   * @return the URI identifier for the default digest algorithm (MD5)
   */
  public static final URI getDefaultAlgorithmId() {
    // Create a URI to return
    return URI.create(ALG_URI_PREFIX + MD5);
  }

  /**
   * @return the String name of the default digest algorithm (MD5)
   */
  public static final String getDefaultAlgorithmName() {
    return MD5;
  }
```

```
}
```

The class will really be required when it comes to handling multiple digest algorithms.

6.5.3. Digest Algorithm Identification

First we create an instance of MessageDigest that will calculate an MD5 checksum. In Java this is done by calling a static factory method:

```
java.security.MessageDigest.getInstance(String algorithm)
```

The String parameter selects the particular algorithm. This is a little messy as if the String passed is invalid the method throws a NoSuchAlgorithmException if the String passed is invalid. For the initial implementation only the identifier for the MD5 algorithm is required, MessageDigest expects the String "MD5" for this algorithm.

There is also a wider issue here. I did not wish to tie the way that the algorithms identified within Planets to Java implementation details. Generally within Planets we have used URIs for identifiers. With this in mind I decided to use URIs of the form **planets:digest/alg/*identifier*** to identify algorithms. The URI for the MD5 algorithm would be **planets:digest/alg/MD5**. If the JavaDigestUtils class in 6.5.2 is re-examined it is clear how this handles both the Java identifier and the Planets URI identifier. This idea will be expanded when support for multiple algorithms is added to the service.

6.5.4. Calculating the MD5 Checksum

First we need to get an instance of MessageDigest:

```
// OK let's try to get the digest algorithm
MessageDigest messDigest =
    MessageDigest.getInstance(JavaDigestUtils.getDefaultAlgorithmName());
```

This uses the JavaDigestUtils method to obtain the identifier of the default algorithm.

Then we need a java.io.InputStream for the content byte sequence of the DigitalObject passed to the method. The Content object provides a getInputStream method that can be called directly:

```
// Now calc the result, we need the bytes from the object
// so let's get the stream
InputStream inStream = digitalObject.getContent().getInputStream();
```

Next we calculate the digest. The digest is calculated from the InputStream. A private utility method in the JavaDigest class handles this process:

```
/**
 * Feeds an input stream to the digest algorithm in chunks of the
 * requested size
 *
 * @param messDigest the java.security.MessageDigest checksum algorithm
 * @param inStream the java.io.InputStream containing the byte sequence
 *        to be added to the digest
 * @param chunkSize the size of the chunks to be fed to the digest
 *        algorithm in bytes, i.e. 1024 = 1KB chunks.
 * @return the total number of bytes in the stream
 * @throws IOException when there's a problem reading from the
 *         InputStream inStream
 */
private int addStreamBytesToDigest(MessageDigest messDigest,
                                   InputStream inStream,
                                   int chunkSize) throws IOException {
  // Save the total number of bytes added to digest for return
  int totalBytes = 0;

  // byte[] for file reading / digest feeding
  byte[] dataBytes = new byte[chunkSize];

  // First read
  int numRead = inStream.read(dataBytes);

  // Now loop through the rest of the file
  while (numRead > 0) {
    // Feed the chunk to the digest algorithm
    messDigest.update(dataBytes, 0, numRead);
    totalBytes += numRead;

    // Get the next chunk
    numRead = inStream.read(dataBytes);
  }

  // Return total bytes read
  return totalBytes;
}
```

This method is then called from within the calculateChecksum method. It forms part of an if statement which catches a trick case, when the object passed has an empty byte sequence associated with it:

```
// Catch the special case of no data in the file
if (this.addStreamBytesToDigest(messDigest,
                                inStream,
                                JavaDigest.DEFAULT_CHUNK_SIZE) < 1)
```

The JavaDigest.DEFAULT_CHUNK_SIZE is a class static and dictates the size, in KB, of the chunks that the digest calculation code reads from the stream. This is set to 1024 KB chunks presently.

If this completes successfully the ServiceReport and FixityResult are populated for return to the caller:

```
// OK, success so create the result
retReport = new ServiceReport(ServiceReport.Type.INFO,
                              ServiceReport.Status.SUCCESS,
                              JavaDigest.SUCCESS_MESSAGE);

// And wrap it in the result
retResult =
    new FixityResult(JavaDigestUtils.getDefaultAlgorithmId().toString(),
                     messDigest.getProvider().getName(),
                     messDigest.digest(),
                     null,
                     retReport);
```

This populates the FixityResult with the algorithm URI (as a string), the name of the algorithm provider, and the digest byte array. The null represents the properties, these aren't implemented.

The above is nearly complete as a basic MD5 calculation service, there is one more topic, handling unexpected conditions and exceptions.

6.6. Adding Exception Handling

There are only three exception conditions to be handled in our basic service, two of these are thrown by java functions:

- NoSuchAlgorithmException, this is thrown by the MessageDigest.getInstance method if it doesn't recognise the String algorithm identifier.
- IOException, this is thrown by the code that reads from the InputStream in the addStreamBytesToDigest method. This method doesn't handle the exception, it simply throws it to the caller.

The third condititon is one that we detect in code:

- The byte sequence in the DigitalObject passed by the caller is empty.

These exceptions are handled as follows.

6.6.1. NoSuchAlgorithmException

This shouldn't happen as Java comes with the MD5 algorithm built in. Given that I'm intending to enhance the service to allow the user to select the algorithm we should handle this gracefully. In the calculateChecksum method there is a try catch block. The code for handling the algorithm exception is as follows:

```
catch (NoSuchAlgorithmException e) {
  // This shouldn't happen at the moment, Java supports MD5
  // Create the Error ServiceReport
  retReport = new ServiceReport(ServiceReport.Type.ERROR,
                                ServiceReport.Status.TOOL_ERROR,
                                e.getMessage() + " for algorithm " +
                                JavaDigestUtils.getDefaultAlgorithmId() +
                                ".");
  // And wrap it in the result
  retResult = new FixityResult(retReport);
}
```

This adds the error details ot the ServiceReport and then wraps the report in an empty FixityResult. This is returned by the method to the caller.

6.6.2. IOException

This exception is thrown when there is a problem reading the data passed by the caller, specifically the byte sequence wrapped in the Content object contained in the passed DigitalObject. The exception handling is very similar to that for the NoSuchAlgorithmException:

```
catch (IOException e) {
  // OK, a problem reading the file
  // Create the Error ServiceReport
  retReport = new ServiceReport(ServiceReport.Type.ERROR,
                                ServiceReport.Status.TOOL_ERROR,
                                e.getMessage());
  // And wrap it in the result
  retResult = new FixityResult(retReport);
}
```

6.7. Unit Testing

Now we'll show some basic unit tests for our service. This document is not meant to be a tutorial on test driven development. Suffice to say that ideally:

- Every method should have its own unit test.
- The tests should be developed first, then the methods to satisfy the tests.

The web carries a wealth of documentation on this subject, the Junit website, http://www.junit.org/, is a good starting point for Java developers. Indeed the examples in this document are based upon JUnit.

For purposes of this document we're going to test the two main areas of functionality for our service, the provision of the service description, and the calculation of MD5 checksums.

6.7.1. Unit Testing structure

The source code for the unit tests lives under our pre-defined test source area:

```
/pserv/PC/javadigest/src/test/java
```

The code must be packaged properly under here, our tests will be put in the

```
eu.planets_project.ifr.core.services.fixity.javadigest
```

Package.

The project is a simple one, so the test organisation doesn't need to be over the top, even so it's good practice to write small test files for a project and co-ordinate them in a test suite. Our test suite is simple:

```
package eu.planets_project.ifr.core.services.fixity.javadigest;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

/**
 * Suite to run all tests in the JavaDigest service project.
* @author <a href="mailto:carl.wilson@bl.uk">Carl Wilson</a>
 */

@RunWith(Suite.class)
@Suite.SuiteClasses( { JavaDigestTests.class })
public class AllJavaDigestSuite { }
```

It only runs one test class, JavaDigestTests, which is where all of our tests will live. The initial set up for this class simply creates a test class and a static instance of the service for testing. The skeleton test file is below:

```
package eu.planets_project.ifr.core.services.fixity.javadigest;

import static org.junit.Assert.fail;

import org.junit.BeforeClass;

import eu.planets_project.services.fixity.Fixity;
```

```
import eu.planets_project.services.utils.test.ServiceCreator;

/**
 * @author <a href="mailto:carl.wilson@bl.uk">Carl Wilson</a>
 */
public class JavaDigestTests {

  // A static instance of a Fixity interface object for testing
  static Fixity javaDigest = null;

  /**
   * Tests JavaDigest Checksum creation
   */
  @BeforeClass
  public static void setup() {
    javaDigest =
        ServiceCreator.createTestService(Fixity.QNAME,
             JavaDigest.class,
             "/pserv-pc-javadigest/javaDigest?wsdl");
  }
  /**
   * Test method for {@link
eu.planets_project.ifr.core.services.fixity.javadigest.JavaDigest#describ
e()}.
   */
  @Test
  public void testDescribe() {
    fail("Not implemented");
  }
}
```

The only test method is for the describe() method, but this is not implemented.

The setup() routine is run onces before the instantiation of the class (see JUnit documenation as to how this is achieved). It uses the Planets:

```
import eu.planets_project.services.utils.test.ServiceCreator;
```

class. This works out wether the unit tests are running in one of the three modes and takes the following action:

- Local – The interface object is just instantiated and returned, this is truly local testing with no web services involved.
- Standalone – In this case a lightweight Jetty web server is used to deploy the service and it is tested as a web service.
- Server – This requires a running Planets instance and tests against the service deployed in a production Planets environment.

6.7.2. Testing describe()

The first test is pretty basic, we'd like to test describe, but there's not that many sensible tests. The simplest idea is to make sure that describe doesn't return a null object.

This simple test looks as follows:

```
  @Test
  public void testDescribe() {
    ServiceDescription desc = javaDigest.describe();
    assertNotNull("The ServiceDescription should not be NULL.", desc);
    System.out.println("Recieved service description: " +
        desc.toXmlFormatted());
  }
```

More sophisticated tests are possible but depend upon the implementation, it is left to the reader to think of better tests when implementing their own service.

6.7.3. Testing MD5 Digest Creation

The meat of the testing is to make sure that the service creates genuine MD5 digests and that they're correct in binary and String form. The best way to do this is to use a third party implementation of the MD5 algorithm. This can be run against a bytes stream, as well as the service, and the results compared. If the digest is identical in all cases it's likely that the service is functioning correctly.

The Apache Commons Codec project provides multiple, alternate digest algorithm implementations, including MD5. The latest 1.4 jar was downloaded from http://commons.apache.org/codec and included in the lib folder for the javadigest project.

The pserv project supplies a TestFile class in:

```
eu.planets_project.services.utils.test.TestFile
```

that allows a user to conveniently obtain files of specific types for testing purposes.

The intention is to test the digest algorithm on many files so the underlying test routines are written as private methods that will be called by the test methods. First we'll develop the private test methods. There are two parts to each test, to test the value of the digest and then the contents of the returned FixityResult / ServiceReport.

The routine to test the digest takes two parameters, a TestFile and a Fixity interface (our service):

```
private void testDefaultDigest(TestFile testFile, Fixity fixity)
```

First the method calls the calculateChecksum() method offered by the Fixity interface passed:

```
  // Ok let's make the call to test
  FixityResult fixityResult =
      fixity.calculateChecksum(
          new DigitalObject.Builder(
            Content.byReference(new
                File(testFile.getLocation()))).build(), null);
```

Then the Apache Commons Codec MD5 routine is used to generate another byte array hash for comparison:

```
  // Use the apache codec MD5 algorithm
  File theFile = new File(testFile.getLocation());
  InputStream inStream = new FileInputStream(theFile);
  byte[] hash = DigestUtils.md5(inStream);
  inStream.close();

  // Assert that the hashes are equal
  assertTrue("Expecting Fast MD5 and Java MD5 byte hashes to be equal",
             Arrays.equals(hash, fixityResult.getDigestValue()));
```

Note the use of assertTrue and the Arrays.equals(byte[], byte[]) method to perform the test. This is because direct comparison of byte arrays fails even when the contents are equal as it compares the Array objects themselves.

Finally the hex string values of the digest are compared for reference, and to test the FixityResult.getDigestValueAsString() method:

```
  // Check the hex string value of the hash
  InputStream newStream = new FileInputStream(theFile);
  String hexhash = DigestUtils.md5Hex(newStream);

  // Assert that the string hashes are equal
  assertEquals("Expecting string hashes to be equal",
               hexhash,
               fixityResult.getDigestValueAsString());
```

Finally the checkResult(FixityResult fixityResult) method is called to test the result. This method just performs some basic checks on the FixityResult:

- Ensure that the ServiceReport type is INFO.

- Ensure that the ServiceReport status is SUCCESS.
- Make sure that the digest value, the hex string digest value, the algorithm id and the provider name are not null/

The implementation is pretty straight forward:

```
private void checkResult(FixityResult fixityResult) {
  if (fixityResult.getReport().getType() != ServiceReport.Type.INFO) {
    System.out.println("Problem with Service Report");
    System.out.println(fixityResult.getReport().getMessage());
  }
  // We'd expect the Report type to be INFO
  assertEquals("Expected ServiceReport.Type to be " +
                            ServiceReport.Type.INFO,
              ServiceReport.Type.INFO,
              fixityResult.getReport().getType());

    // We'd expect the Report status to be SUCCESS
    assertEquals("Expected ServiceReport.Status to be " +
                           ServiceReport.Status.SUCCESS,
                fixityResult.getReport().getStatus(),
                ServiceReport.Status.SUCCESS);
  // We'd not expect a null Digest value or
  // algorithm identifier, or provider
  assertNotNull("FixityResult.getDigestValue() should not be null",
                fixityResult.getDigestValue());
  assertNotNull("FixityResult.getDigestValueAsString() " +
                                     "should not be null",
                fixityResult.getDigestValueAsString());
  assertNotNull("FixityResult.getAlgorithmId() should not be null",
                fixityResult.getAlgorithmId());
  assertNotNull("FixityResult.getAlgorithmprovider() should not be null",
                fixityResult.getAlgorithmProvider());
}
```

All that we now require is to add a test for each type of test file we want to use. The one downside of the approach is that every file requires its own test, we won't show every test line, the first three establish the procedure:

```
/*
 * To get more informative test reports, we wrap every enum element
 * into its own test. We could iterate over the enum elements instead
 * (see below).
*/
@Test public void testRtf(){testDefaultDigest(TestFile.RTF, javaDigest);}
@Test public void testBmp(){testDefaultDigest(TestFile.BMP, javaDigest);}
@Test public void testXml(){testDefaultDigest(TestFile.XML, javaDigest);}
```

6.8. Adding Support for Multiple Digest Algorithms

We now have a working service with a set of unit tests. The final enhancement we'll look add is the addition of a single user parameter. MD5 is just one of several digest algorithms provided by the Java MessageDigest class The idea is to allow the user to choose between all algorithms provided by their Java environment.

First we'll ammend the Service Description so that it carries information about the parameter. The description class holds a List of Parameter objects:

```
eu.planets_project.services.datatypes.Parameter
```

The parameters that a service takes are always implementation specific. In this case I wanted the caller to be able to specify the name of the digest algorithm to be invoked. The Java MessageDigest class uses String names to identify the algorithm to be used, e.g. MD5, SHA256. I

wanted something slightly less ambiguous so decided to use a set of URI identifiers that I could map to the Java names. In the utility class

```
eu.planets_project.ifr.core.services.fixity.javadigest.JavaDigestUtils
```

I added a URI prefix to place in front of the Java algorithm name:

```
private static final String ALG_URI_PREFIX =
         “planets:digest/alg/”;
```

I then just tag the Java name onto the end of the URI.

Within the

```
eu.planets_project.ifr.core.services.fixity.javadigest.utils
```

package there is the JavaDigestDescription class used to create the Service Description in 6.4. The class needs an additional private method to add the Parameter options to the service description. First add a few static String constants to the top of the class

```
/** The name of the parameter for the algorithm ID */
public static final String ALG_PARAM_NAME = "AlgorithmId";
/** The type of the parameter for the algorithm ID */
public static final String ALG_PARAM_TYPE = "URI";
/** The description of the parameter for the algorithm ID */
private static final String ALG_PARAM_DESC =
    "A Planets digest algorithm URI identifying the " +
    "requested algorithm, supported values: ";
private static final String LIST_SEP = ", ";
```

For convenience I added the parameter creation code in a private getParameters() method. This method returns a list of Parameter objects that can be used by the describe() method to add to the service description.

```
/**
 * A private helper method that puts together the java.util.List of
 * eu.planets_project.services.datatypes.Parameter objects for the
 * eu.planets_project.services.datatypes.ServiceDescription.
 *
 * These are the parameters taken by the fixity service
 *
 * @return the java.util.List of parameters
 */
private static List<Parameter> getParameters() {
  List<Parameter> paramList = new ArrayList<Parameter>();
  // Add the algorithm selection parameter from a builder
  // We need the name and the default value
  Parameter.Builder algBuilder =
    new Parameter.Builder(JavaDigestDescription.ALG_PARAM_NAME,
            JavaDigestUtils.getDefaultAlgorithmId().toString());

  // We need a description, the prefix is OK
  String algParamDesc = JavaDigestDescription.ALG_PARAM_DESC;

  // Get the algs from the utils
  for (URI uri : JavaDigestUtils.getDigestAlgorithmIds()) {
    // And add one for each alg plus a list separator
    algParamDesc += uri + JavaDigestDescription.LIST_SEP;
  }
  // We can now add the description
  // but we'll need to chop off the last list separator
  algBuilder.description(algParamDesc.substring(0,
                                        algParamDesc.length() -
                         JavaDigestDescription.LIST_SEP.length()));
  // Finally the type and deliver parameter goodness to our list
  algBuilder.type(JavaDigestDescription.ALG_PARAM_TYPE);
```

```
  paramList.add(algBuilder.build());

  // Return an unmodifiable list of parameters
  return Collections.unmodifiableList(paramList);
}
```

The parameter method calls a utilility method JavaDigestUtils.getDigestAlgorithmIds(), this simply returns the full set of the digest algorithm URI identifiers. The implementation details aren't covered in this tutorial.

Finally add a call to the getDescription() method, just before the service description is built and returned to add the Parameter list to the builder:

```
// Last is worst, the parameters
sd.parameters(JavaDigestDescription.getParameters());

// Return the description
return sd.build();
```

Now all that remains to do is to add the code to the service class itself to process any passed Parameters, check the requested algorithm and obtain it. A private method to check the parameter list and set the algorithm is required. This should check the passed parameter list, locate any that specify the algorithm name and then return the requested algorithm, or a default. First a static constant at the top of the JavaDigest class to deal with the no such algorithm case:

```
private static final String NO_ALG_MESSAGE = "The MessageDigest " +
            "function does not implement the algorithm ";
```

Then the method to check the Parameter list and return the algorithm URI:

```
private URI getDigestIdFromParameters(List<Parameter> params)
    throws NoSuchAlgorithmException, URISyntaxException {
  // The return value, set to default
  URI retVal = JavaDigestUtils.getDefaultAlgorithmId();

  // Now check out that parameter list, if it's not null
  if (params != null) {
    for (Parameter param : params) {
      // It's the algorithm identifier param
      if (param.getName().equals(JavaDigestDescription.ALG_PARAM_NAME)) {
        try {
          // If it's a valid alg id
          if (JavaDigestUtils.hasAlgorithmById(
                              URI.create(param.getValue())))
          // It's an OK algorithm return it
          return URI.create(param.getValue());
          // It's not a valid algorithm ID so throw
          throw new NoSuchAlgorithmException(NO_ALG_MESSAGE +
                                             param.getValue());
        } catch (IllegalArgumentException e) {
          // OK the URI has blown so throw the underlying cause
          throw (URISyntaxException)e.getCause();
        }
      }
    }
  }
  return retVal;
}
```

A utility method is added to the JavaDigestUtils that given an algorithm identifier URI, returns the Java string name of the algorithm. This can then be used at the start of the JavaDigest.calculateChecksum class to obtain the appropriate MessageDigest algorithm:

```
// Let's get the requested message digest from the params (or default)
URI requestedAlgId = this.getDigestIdFromParameters(parameters);

// OK let's try to get the digest algorithm
MessageDigest messDigest = MessageDigest.getInstance(
           JavaDigestUtils.getJavaAlgorithmName(requestedAlgId));
```

Additional exception handling and tests are not included in this tutorial.

7. Xena – An Almost Pure Java Service

The idea of the Xena Preservation Service was to add Open Office Document support to Planets by leveraging the work done on the Xena Project by National Archives of Australia. After inspecting the source, it was found that the OfficeToXenaOooNormaliser.java class provided a good example of how to use Open Office from Java in order to translate file formats.

Based on this approach, the project was laid out as shown in in the repository and started working on a eu.planets_project.services.migration.xenaservice.XenaMigrations.java class that could handle the general case of migrations using this approach. I then added service classes to the Xena services, by looking at the BasicMigrateOneBinary.java interface class and copying that into a new DocToODFXena.java class. Note that the package names do not have to have any particular form.

This new class implements the BasicMigrateOneBinary interface, and has a copy of all of the web service annotations needed by this service. However, it is very important to note that the WebMethod annotation declares a unique Name of 'DocToODFXena', and a standard serviceName of 'BasicMigrateOneBinary'.

I then added the standard EJB annotations, e.g. `@Stateless`. Adding this ensure that the application server attempts to deploy the bean and the web services too.

As Xena requires a working installation of Open Office, I added a properties file to the project resources, called /eu/planets_project/services/migration/xenaservices/xena.properties. This defines a property pointing to my Open Office installation, and is read by my XenaMigrations class on initialization. I then added a README.txt file to describe this configuration procedure to other users.

The body of the DocToODFXena.java is very simple, and just configures the XenaMigrations class for the required conversion and then invokes it. The only other requirement is to add the xena-4.1-office.jar to the PA/xena/lib directory.

The provided build script is invoked with ant, and this combines the compiled classes, the resources, and the libraries into a single EJB jar that can then be deployed to the application server. The various client programs provided by the Planets project can then be used to invoke the services.

8. A Roadmap for the Service Interfaces

8.1. Definition of New Service Interfaces

Currently, we've only been able to standardise interfaces for a few basic preservation services. Documentation for the current set of interfaces can be found on the Planets Wiki Service Interface page. New service categories will be required, some are already anticipated. As an example there is a category for comparator services, it is anticipated these will be used to compare items such as binaries, metadata, etc. but there has been no formal attempt to define interfaces or describe methods yet. Preservation planning services are in a similar state.

Service types considered candidates for new interface definitions currently include:

- Compare Binaries
- Compare Metadata
- Encrypt/Decrypt/Sign/Stamp/Watermark
- Make Anonymous/Clean/Optimise/Compress?/Repair?
- Component Discovery & Extraction
- Relocate - move between data registries
- Crawl - Receive a URL and crawl it

Although the first steps have made it possible to bring a lot of preservation tools into the Planets framework, we will need to agree interfaces and standards for more complex and powerful services in the future.

8.2. Definition of New Service Categories

As has already been pointed out, the list of service categories is not intended to be complete.

In addition we expect to have to define new categories to support new functionality that had not been anticipated.

8.3. Complex Digital Object Types

There is a requirement for more sophisticated Digital Object types, e.g.

- Resource References (like URLs)
  - Migrate a JDBC URI to a SIARD file.
- Compound Formats
  - A directory of files
  - A sequence of images (for rendered pages)
  - Other container formats?

8.4. Large File Issues

The current Digital Object implementation and service interface annotations will allow MOTM and data streaming. This means that large files can be transferred between web services running on Java Virtual Machines with small amounts of memory.

Further work to test the transfer of large amounts binary data is currently underway. New utilitlites may be added and implementation classes will be enhanced if issues are discovered and can be fixed.

8.5. Enhancements for Batching and Asynchronous Service Support

Planets services are currently simple, low level and atomic. One of their features is they only handle single files. There is an overhead associated with making web service calls and it is obviously inefficient to call the same service 100 times as opposed to calling it once and passing 100 files.

This will also have implications for the way in which services are invoked. Currently all calls are synchronous, but large batches will cause long waits and possible time outs or issues with connection reliability.

See

- Asynchronous Web Service Invocation with JAX-WS 2.0.
- JBossWS User Guide - Asynchronous Invocations.

One idea is the development of an asynchronous service that implements a batch handling mechanism. The caller would submit the batch, and synchronously receive a 'ticket' that can then be used to poll the service on a known interface to enquire as to the jobs completion. When the job is complete the polling caller would be passed the results or a reference to them, for instance a URL. This could be an interface, with a reference implementation that would provide Java developers with a lot of the code required.

8.6. Secure Web Services

Currently all of the services are called anonymously with access only restricted by local infrastructure, i.e. firewalls, etc..The Web Services specifications address ways of handling authentication and authorisation across interoperable web services.

See:

- JBossWS User Guide - Authentication
- JBossWS User Guide - Secure Transport.

9. Getting Involved

The IF project cannot do this alone, as the members of that project do not accurately represent the intended users of these interfaces.In particular, we need to ensure that our service types can encapsulate the needs of the PA and PC teams.If you want to be involved in creating the next generation of Planets services, please join the techie@planets-project.eu mailing list and send an email describing the kind of functionality you would like to see added to Planets.

Currently, our 'agreed' services are shown on the Tech/Planets Preservation Services wiki page, although the authoritative definitions of these services will always be the Java versions found in the SVN repository, here: http://gforge.planets-project.eu/svn/if_sp/trunk/components/common/src/main/java/eu/planets_project/ifr/core/common/services/.

Our current proposed services are shown on the Tech/Planets Preservation Services (Dev) wiki page. This latter page is rough in places, but contains many ideas for future service types and is the place where the future service definitions will be placed and discussed, alongside any discussions on the mailing lists.

There are lots of other things we'd like services for, and lots of tools we would like to wrap. See the following pages for inspiration:

- Wrapped Tools Roadmap
- Migration Tools
- Existing tools for information extraction transformation and management
- Emulation Tools

Note that pure Java tools have a distinct advantage in terms of ease of deployment.

9.1. Software That Might be Wrapped

This list should be merged with the Migration Tools document. See also Idea for Integrating Existing Software As Services.

- Standard Java - tools included in the JRE as of 1.5, or standard pure-Java add ons:
  - ImageIO: http://java.sun.com/j2se/1.5.0/docs/api/javax/imageio/package-summary.html#package_description
  - JIMI from Sun: http://java.sun.com/products/jimi/
  - http://schmidt.devlib.org/java/image-faq/read-write-image-files.html
- Batik (SVG).
- FOP.
- XML Graphics Commons.

Apache POI - Java API To Access Microsoft Format Files.

Appendix A Service Interfaces

A.1 Introduction

This section provides:

- Some general guidelines for service implementers
- Some additional information for specific interfaces and a code snapshot for each interface

A.2 General Guidelines for Implementors

The following information applies to the development of all, or at least most, categories of Planets Service. The specific sections that follow contain information about specific interfaces.

A.2.1 Returning a DigitalObject

Any service that returns a DigitalObject should return a new instance containing the new binary (by value or reference). A service shouldn't attempt to clone / copy the details from the DigitalObject passed to it, that is a decision for the caller, at the workflow / institutional level. This separates the service developer from concerns as to a particular caller holds / serializes their digital objects and metadata.

A.2.2 Use of Parameters

All services can take a list of parameters to allow the developer to implement richer functionality if desired. An example might be to control the compression algorithms or levels for a format migration service.

A.2.3 Default Behavior and Parameters

Every service should document its default behavior in its Service Description. This should be the behavior when:

- The caller supplies a null list of Parameter objects.
- The caller supplies an empty list of Parameter objects.

A.3 PlanetsService

Every Planets Service Interface extends the PlanetsService interface, which provides common behavior for all Planets interfaces.

A.3.1 The Source Code

```
/**
 *
 */
package eu.planets_project.services;

import javax.jws.WebMethod;
import javax.jws.WebResult;
import javax.xml.ws.ResponseWrapper;

import eu.planets_project.services.datatypes.ServiceDescription;

/**
 * @author <a href="mailto:Andrew.Jackson@bl.uk">Andy Jackson</a>,
 *         <a href="mailto:fabian.steeg@uni-koeln.de)>Fabian Steeg</a>
 */
public interface PlanetsService {
  String NAME = "PlanetsService";

  /**
   * A method that can be used to recover a rich service description,
   * and thus populate a service registry.
   * @return A ServiceDescription object that describes this service,
   *         to aid service discovery.
   */
  @WebMethod(operationName = PlanetsService.NAME + "Describe",
             action = PlanetsServices.NS
             + "/" + PlanetsService.NAME + "/" + "Describe")
  @WebResult(name = PlanetsService.NAME + "Description",
             targetNamespace = PlanetsServices.NS
             + "/" + PlanetsService.NAME, partName = PlanetsService.NAME
             + "Description")
  @ResponseWrapper(className = "eu.planets_project.services."
             + PlanetsService.NAME + "DescribeResponse")
  ServiceDescription describe();
}
```

A.4 The Fixity Interface

A.4.1 Fixity Interface Source Code

```
/**
 *
 */
package eu.planets_project.services.fixity;

import java.util.List;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingType;

import eu.planets_project.services.PlanetsService;
import eu.planets_project.services.PlanetsServices;
import eu.planets_project.services.datatypes.DigitalObject;
import eu.planets_project.services.datatypes.Parameter;

/**
 * @author CFWilson
*/
@WebService(name = Fixity.NAME, targetNamespace = PlanetsServices.NS)
@BindingType(value =
             "http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true")
public interface Fixity extends PlanetsService {
  /** The interface name */
  String NAME = "Fixity";
  /** The qualified name */
  QName QNAME = new QName(PlanetsServices.NS, Fixity.NAME);

  /**
   * @param digitalObject The Digital Object to be identified.
   * @param parameters
   * @return Returns a Types object containing the identification result
   */
  @WebMethod(operationName = Fixity.NAME,
             action = PlanetsServices.NS + "/" + Fixity.NAME)
  @WebResult(name = Fixity.NAME + "Result",
             targetNamespace = PlanetsServices.NS + "/" + Fixity.NAME,
             partName = Fixity.NAME + "Result")
  FixityResult calculateChecksum(
      @WebParam(name = "digitalObject",
                targetNamespace = PlanetsServices.NS + "/" + Fixity.NAME,
                partName = "digitalObject")
        DigitalObject digitalObject,
      @WebParam(name = "parameters",
                targetNamespace = PlanetsServices.NS + "/" + Fixity.NAME,
                partName = "parameters")
        List<Parameter> parameters);
}
```

A.4.2 The FixityResult Source Code

```
/**
 *
 */
package eu.planets_project.services.fixity;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;

import eu.planets_project.services.datatypes.Property;
import eu.planets_project.services.datatypes.ServiceReport;

/**
 * @author CFWilson
 */
@XmlRootElement
@XmlAccessorType(value = XmlAccessType.FIELD)
public class FixityResult {

  private String algorithmId;
  private byte[] digestValue;
  private List<Property> properties;
  private ServiceReport report;

  /**
   * No-arg constructor required by JAXB
   */
  @SuppressWarnings("unused")
  private FixityResult(){/** NULL & private to prevent no arg
construction */};

  /**
   * @param algorithmId
   *        the java.lang.String identifier of the digest algorithm used
   * @param digestValue
   *        the byte[] digest value
   * @param properties
   *        any service properties to add to the report
   * @param report
   *        the service report
   */
  public FixityResult(String algorithmId,
                      byte[] digestValue,
                      List<Property> properties,
                      ServiceReport report) {
    this.algorithmId = algorithmId;
    this.digestValue = digestValue.clone();
    this.properties = properties;
    this.report = report;
  }

  /**
   * @param algorithmId
   *        the java.lang.String identifier of the digest algorithm used
   * @param properties
   *        any service properties to add to the report
```

```
   * @param report
   *        the service report
   */
  public FixityResult(String algorithmId,
                      List<Property> properties,
                      ServiceReport report) {
    this.algorithmId = algorithmId;
    this.digestValue = null;
    this.properties = properties;
    this.report = report;
  }

  /**
   * @param properties
   *        any service properties to add to the report
   * @param report
   *        the service report
   */
  public FixityResult(List<Property> properties, ServiceReport report) {
    this.algorithmId = null;
    this.digestValue = null;
    this.properties = properties;
    this.report = report;
  }

  /**
   * @param report
   *        the service report
   */
  public FixityResult(ServiceReport report) {
    this.algorithmId = null;
    this.digestValue = null;
    this.properties = null;
    this.report = report;
  }

  /**
   * @return the java.lang.String algorithm ID
   */
  public String getAlgorithmId() {
       return this.algorithmId;
  }

  /**
   * @return the digest value as a java.lang.String
   */
  public String getDigestValueAsString() {
    return this.digestValue.toString();
  }

  /**
   * @return the digest values as a byte array
   */
  public byte[] getDigestValue() {
    return this.digestValue;
  }

  /**
   * @return the an unmodifiable
   *     java.util.List<eu.planets_project.services.datatypes.Property>
   *     of properties associated with the FixityResult
   */
  public List<Property> getProperties() {
```

```
    return this.properties == null ?
        // If propertie
        Collections.unmodifiableList(new ArrayList<Property>()) :
        Collections.unmodifiableList(this.properties);
  }

  /**
   * @return
   *    the eu.planets_project.services.datatypes.ServiceReport report
   */
  public ServiceReport getReport() {
    return this.report;
  }
}
```

A.5 The Identify Interface

A.5.1 The Identify Interface Source Code

```
/**
 *
 */
package eu.planets_project.services.identify;

import java.util.List;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingType;

import eu.planets_project.services.PlanetsService;
import eu.planets_project.services.PlanetsServices;
import eu.planets_project.services.datatypes.DigitalObject;
import eu.planets_project.services.datatypes.Parameter;

/**
 * Identification of a DigitalObject, returning a types object
 * containing the identified Pronom URIs and the status resulting
 * from the identification.
 * @author Fabian Steeg, Andrew Jackson
 */
@WebService(name = Identify.NAME, targetNamespace = PlanetsServices.NS)
@BindingType(value =
    "http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true")
public interface Identify extends PlanetsService {

  /** The interface name */
  String NAME = "Identify";
  /** The qualified name */
  QName QNAME = new QName(PlanetsServices.NS, Identify.NAME);

  /**
   * @param digitalObject The Digital Object to be identified.
   * @return Returns a Types object containing the identification result
   */
  @WebMethod(operationName = Identify.NAME,
             action = PlanetsServices.NS + "/" + Identify.NAME)
  @WebResult(name = Identify.NAME + "Result",
             targetNamespace = PlanetsServices.NS
            + "/" + Identify.NAME, partName = Identify.NAME + "Result")
  IdentifyResult identify(@WebParam(name = "digitalObject",
                                    targetNamespace = PlanetsServices.NS
                                    + "/" + Identify.NAME,
                                    partName = "digitalObject")
                                   DigitalObject digitalObject,
                          @WebParam(name = "parameters",
                                    targetNamespace = PlanetsServices.NS
                                    + "/" + Identify.NAME
                                    partName = "parameters")
                                    List<Parameter> parameters);
}
```

A.5.2 The IdentifyResult Source Code

```
/**
 *
 */
package eu.planets_project.services.identify;

import java.net.URI;
import java.util.ArrayList;
import java.util.List;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;

import eu.planets_project.services.datatypes.Property;
import eu.planets_project.services.datatypes.ServiceReport;

/**
 * Result type for identification services.
 * @author <a href="mailto:Andrew.Jackson@bl.uk">Andy Jackson</a>
 */
@XmlRootElement
@XmlAccessorType(value = XmlAccessType.FIELD)
public final class IdentifyResult {

  private List<URI> types;
  private Method method;
  private List<Property> properties;
  private ServiceReport report;

  /**
   * The Method enumeration is intended to allow you state what kind of
   * evidence was used to create this identification result.
   */
  public enum Method {

    /**
     * 'Extension' means that this identification was based purely on
     * the filename extension of the supplied digital object.
     */
    EXTENSION,
    /**
     * 'Metadata' means that this identification was based on
     *other metadata supplied with the digital object,
     * but the objects bytes were not inspected.
     */
    METADATA,
    /**
     * 'Magic' means that this identification was based on matching the
     * digital object bytes against a bytestream signature,
     * a.k.a. a 'file
     * Magic number'. In this case, a small set of bytes from the
     * file has been inspected, and in general the inspection does not
     * depend on the file size.
     */
    MAGIC,
    /**
     * 'Partial Parse' means that this identification was based on a
     * detailed inspection of the digital object bytestreams, and a
     * significant proportion of the objects bytes were used to
     * reach this conclusion.
```

```
     */
    PARTIAL_PARSE,
    /**
     * 'Full Parse' means that this identification was based on a
     * detailed inspection of the digital object bytestream, and
     * every single bit was 'touched'.
     */
    FULL_PARSE,
    /**
     * Means that this identification was based on some other evidence.
     * Please contact the IF so that more Methods can be added.
     */
    OTHER
  };

  /**
   * No-args constructor required by JAXB.
   */
  @SuppressWarnings("unused")
  private IdentifyResult() {}

  /**
   * @param types The list of matching format URIs,
   *              specifying the formats the object is consistent with.
   * @param method The method used to reach this decision.
   *               Should not be set to NULL unless completely
   *               unavoidable.
   * @param report The standard service report object.
   * @param properties Any other properties determined by the service.
   */
  public IdentifyResult(final List<URI> types,
                        final Method method,
                        final ServiceReport report,
                        final List<Property> properties) {
    this.types = types;
    this.method = method;
    this.report = report;
    this.properties = properties;
  }

  /**
   * @param types The list of matching format URIs, specifying the
   *              formats the object is consistent with.
   * @param method The method used to reach this decision.
   *               Should not be set to NULL unless completely
   *               unavoidable.
   * @param report The standard service report object.
   */
  public IdentifyResult(final List<URI> types,
                        final Method method,
                        final ServiceReport report) {
    this.types = types;
    this.method = method;
    this.report = report;
    this.properties = null;
  }

  /**
   * @return A copy of the types
   */
  public List<URI> getTypes() {
      return types == null ? new ArrayList<URI>() :
                             new ArrayList<URI>(types);
```

```
  }

  /**
   * @return the method
   */
  public Method getMethod() {
      return method;
  }

  /**
   * @return the report
   */
  public ServiceReport getReport() {
      return report;
  }

  /**
   * @return A copy of the properties
   */
  public List<Property> getProperties() {
    return new ArrayList<Property>(properties);
  }
}
```

A.6 The Validate Interface

A.6.1 The Validate Interface Source Code

```
/**
 *
 */
package eu.planets_project.services.validate;

import java.net.URI;
import java.util.List;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingType;

import eu.planets_project.services.PlanetsService;
import eu.planets_project.services.PlanetsServices;
import eu.planets_project.services.datatypes.DigitalObject;
import eu.planets_project.services.datatypes.Parameter;

/**
 * Validation of a DigitalObject.
 *
 * @author Fabian Steeg, Andrew Jackson, Asger Blekinge-Rasmussen
 */
@WebService(name = Validate.NAME, targetNamespace = PlanetsServices.NS)
@BindingType(value =
             "http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true")
public interface Validate extends PlanetsService {
  /** The interface name */
  public static final String NAME = "Validate";
  /** The qualified name */
  public static final QName QNAME =
      new QName(PlanetsServices.NS, Validate.NAME);

  /**
   * @param digitalObject
   *            The Digital Object to be validated.
   * @param format
   *            The format that digital object purports to be in
   * @param parameters
   *            a list of parameters to provide fine grained tool control
   * @return Returns a ValidateResult object with the result of the
   *         validation
   */
  @WebMethod(operationName = Validate.NAME,
             action = PlanetsServices.NS + "/" + Validate.NAME)
  @WebResult(name = Validate.NAME + "Result",
             targetNamespace = PlanetsServices.NS + "/" + Validate.NAME,
             partName = Validate.NAME + "Result")
  public ValidateResult validate(
          @WebParam(name = "digitalObject",
                    targetNamespace = PlanetsServices.NS + "/" +
                        Validate.NAME,
                    partName = "digitalObject")
            DigitalObject digitalObject,
          @WebParam(name = "format",
                    targetNamespace = PlanetsServices.NS + "/" +
```

```
                        Validate.NAME, partName = "format")
            URI format,
          @WebParam(name = "parameters",
                    targetNamespace = PlanetsServices.NS + "/" +
                      Validate.NAME,
                    partName = "parameters")
            List<Parameter> parameters);
}
```

A.6.2 The ValidateResult Source Code

```
/**
 *
 */
package eu.planets_project.services.validate;

import eu.planets_project.services.datatypes.Property;
import eu.planets_project.services.datatypes.ServiceReport;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;
import java.net.URI;
import java.util.ArrayList;
import java.util.List;

/**
 * Result type for validation services.
 * @author <a href="mailto:Andrew.Jackson@bl.uk">Andy Jackson</a>,
 *         <a href="mailto:abr@statsbiblioteket.dk">
 *         Asger Blekinge-  Rasmussen</a>,
 */
@XmlRootElement
@XmlAccessorType(value = XmlAccessType.FIELD)
public final class ValidateResult {

  /**
   * Validity is a two-step process.
   * First, the file is regarded, to see if it
   * can be parsed as the given format.
   * If it can be thus parsed, ofThisFormat is set to true.
   * If not, false. False means that the file is not really of
   * the specified format. <br/>
   * False correspond to the JHove term of INVALID,
   * which were also used in Planets earlier on. <br>
   * Default true;
   */
  private boolean ofThisFormat;

  /**
   * Only relevant if ofThisFormat is true.
   * When the file have been parsed as
   * of a specific format, errors in regards to that format
   * will be found. <br>
   * A pdf file could be parsed as pdf, but if it lacks certain nessesary
   * datastructures it is not a valid pdf file, but still a pdf file.
   * In that case ofThisFormat would be set to true, but
   * validInRegardsToThisFormat to false. <br/>
   * True corresponds the the JHove term of VALID,
   * which were also used in Planets earlier.
   *  False corresponds to WELL_FORMED. <br>
   * Default true;
   * @see #ofThisFormat
   */
  private boolean validInRegardToThisFormat;

  /**
   * The format that the file was validated against.
   */
  private URI thisFormat;
  /**
```

```
   * The service report for this validation result.
   */
  private ServiceReport report;

  /** Also allow properties to be returned,
   *  to permit extensible behaviour. */
  private List<Property> properties;

  /**
   * The list of errors collected during the validation.
   * Errors are expressed via the contained class Message.
   * A message consist of two strings, adress and description.
   * For this interface, the adress is meant to be a line
   * number. <br>
   * If no line number can be found, or the error is in regards
   * to the entire file, use -1. <br>
   * These errors are tool specific, and no further attempts
   * have been made to standardize them. <br>
   * Errors are problems with the file, that cause either ofThisFormat or
   * validInRegardsToThisFormat to be false.
   * @see #ofThisFormat
   * @see #validInRegardToThisFormat
   * @see #warnings
   * @see eu.planets_project.services.validate.ValidateResult.Message
   */
  private List<Message> errors;

  /**
   * The list of warning collected during the validation.
   * A message consists of two strings, adress and description.
   * For this interface, the adress is meant to be a line number. <br/>
   * If no line number can be found, or the warning is in regards to the
   * entire file, use -1. <br>
   * These warnings are tool specific, and no further attempts
   * have been made to standardize them. <br>
   * Warnings are problems with the file,
   * that are not serious enough to make
   * validInRegardToThisFormat or ofThisFormat false.
   * @see eu.planets_project.services.validate.ValidateResult.Message
   */
  private List<Message> warnings;

  /**
   * No-args constructor required by JAXB.
   */
  private ValidateResult() {}

  /**
   * @param builder The builder to create a validate result from
   */
  private ValidateResult(final Builder builder) {
    errors = builder.errors;
    warnings = builder.warnings;
    ofThisFormat = builder.ofThisFormat;
    validInRegardToThisFormat = builder.validInRegardToThisFormat;
    thisFormat = builder.thisFormat;
    report = builder.report;
    properties = builder.properties;
  }

  /**
   * @return A copy of the collected errors in the file
   * @see #errors
```

```
   */
  public List<Message> getErrors() {
    return new ArrayList<Message>(errors);
  }

  /**
   * @return A copy of the collected warnings
   * @see #errors
   */
  public List<Message> getWarnings() {
    return new ArrayList<Message>(warnings);
  }

  /**
   * @return the report
   */
  public ServiceReport getReport() {
    return report;
  }

  /**
   * @see #ofThisFormat
   * @return whether or not the file was of this format
   */
  public boolean isOfThisFormat() {
    return ofThisFormat;
  }

  /**
   * @see #validInRegardToThisFormat
   * @return whether or not the file was valid in regards to this format
   */
  public boolean isValidInRegardToThisFormat() {
    return validInRegardToThisFormat;
  }

  /**
   * @see #thisFormat
   * @return This format
   */
  public URI getThisFormat() {
    return thisFormat;
  }

  /**
   * @return the properties
   */
  public List<Property> getProperties() {
    return properties;
  }

  /**
   * Messages about errors and warnings collected during the validation.
   * A message consist of two Strings, adress and description.
   */
  public static final class Message {
    /**
     * The adress that contained the data that this message is about.
     * In files, this is just the line-number.
     */
    private String adress;

    /**
```

```
     * The message.
     */
    private String description;

    /** Constructor for JAXB. */
    @SuppressWarnings("unused")
    private Message() {}

    /**
     * Constructor.
     * @param adress The adress of what provoked the message
     * @param description The message
     */
    public Message(final String adress, final String description) {
      this.adress = adress;
      this.description = description;
    }

    /**
     * Construtor.
     * @param description The message
     */
    public Message(final String description) {
      this.adress = "";
      this.description = description;
    }

    /**
     * @return The address
     */
    public String getAddress() {
      return adress;
    }

    /**
     * @return The description
     */
    public String getDescription() {
      return description;
    }

    /**
     * {@inheritDoc}
     * @see java.lang.Object#toString()
     */
    @Override
    public String toString() {
      return "Message{" + "adress='" + adress + '\'' + ", description='"
             + description + '\'' + '}';
    }
  }

  /**
   * Builder for ValidateResult instances.
   */
  public static final class Builder {

    private boolean ofThisFormat;
    private boolean validInRegardToThisFormat;
    private URI thisFormat;
    private ServiceReport report;
    private List<Property> properties;
    private List<Message> errors;
```

```
    private List<Message> warnings;

    /**
     * @param validateResult The instance to create a new instance from
     */
    public Builder(final ValidateResult validateResult) {
      initialize(validateResult);
    }

    /**
     * @param thisFormat The identification result format
     * @param report The identification report
     */
   public Builder(final URI thisFormat, final ServiceReport report) {
      init();
      this.thisFormat = thisFormat;
      this.report = report;
    }

    /**
     * @return The validate result
     */
    public ValidateResult build() {
      return new ValidateResult(this);
    }

    /** Set default values. */
    private void init() {
      ofThisFormat = true;
      validInRegardToThisFormat = true;
      properties = new ArrayList<Property>();
      errors = new ArrayList<Message>();
      warnings = new ArrayList<Message>();
    }

    /**
     * @param validateResult The instance to use for creating
     *                       a new instance
     */
    private void initialize(final ValidateResult validateResult) {
      if (validateResult == null) {
        init();
        return;
      } else {
        ofThisFormat = validateResult.ofThisFormat;
        validInRegardToThisFormat =
            validateResult.validInRegardToThisFormat;
        thisFormat = validateResult.thisFormat;
        report = validateResult.report;
        properties = validateResult.properties;
        errors = validateResult.errors;
        warnings = validateResult.warnings;
      }
    }

    /**
     * @param ofThisFormat See #ofThisFormat
     * @return The builder, for cascaded calls
     */
    public Builder ofThisFormat(final boolean ofThisFormat) {
      this.ofThisFormat = ofThisFormat;
      return this;
    }
```

```
    /**
     * @param validInRegardToThisFormat See #validInRegardToThisFormat
     * @return The builder, for cascaded calls
     */
    public Builder validInRegardToThisFormat(
          final boolean validInRegardToThisFormat) {
      this.validInRegardToThisFormat = validInRegardToThisFormat;
      return this;
    }

    /**
     * @param thisFormat See #thisFormat
     * @return The builder, for cascaded calls
     */
    public Builder thisFormat(final URI thisFormat) {
      this.thisFormat = thisFormat;
      return this;
    }

    /**
     * @param report See #report
     * @return The builder, for cascaded calls
     */
    public Builder report(final ServiceReport report) {
      this.report = report;
      return this;
    }

    /**
     * @param properties See #properties
     * @return The builder, for cascaded calls
     */
    public Builder properties(final List<Property> properties) {
      this.properties = properties;
      return this;
    }

    /**
     * Adds a warning for a specified line number in the file.
     * @param linenumber The offending line
     * @param warning The warning description
     * @return The builder, for cascaded calls
     * @see #warnings
     */
    public Builder addWarning(final int linenumber,
                              final String warning) {
      warnings.add(new Message("line " + linenumber, warning));
      return this;
    }

    /**
     * Adds a warning for line number -1, ie. for the entire file.
     * @param warning the description of the warning
     * @return The builder, for cascaded calls
     * @see #warnings
     */
    public Builder addWarning(final String warning) {
      warnings.add(new Message("line -1", warning));
      return this;
    }

    /**
```

```
     * Adds a error for a specified line number in the file. Remember to
     * also mark the file as not ofThisFormat or not
     * validInRegardToThisFormat
     * @param linenumber the offending line
     * @param error The error description
     * @return The builder, for cascaded calls
     * @see #errors
     * @see #ofThisFormat
     * @see #validInRegardToThisFormat
     */
    public Builder addError(final int linenumber, final String error) {
      errors.add(new Message("line " + linenumber, error));
      return this;
    }

    /**
     * Adds a error for line number -1, ie. for the entire file
     * @param error the description of the error
     * @return The builder, for cascaded calls
     * @see #addError(int, String)
     */
    public Builder addError(final String error) {
      errors.add(new Message("line -1", error));
      return this;
    }
  }
}
```

A.7 The Characterise Interface

A.7.1 The Characterise Interface Source Code

```
package eu.planets_project.services.characterise;

import java.net.URI;
import java.util.List;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingType;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;

import eu.planets_project.services.PlanetsService;
import eu.planets_project.services.PlanetsServices;
import eu.planets_project.services.datatypes.DigitalObject;
import eu.planets_project.services.datatypes.Parameter;
import eu.planets_project.services.datatypes.Property;

/**
 * Characterisation of one digital object. This is the
 * generic characterisation interface for characterisation tools
 * like the XCL Extractor and the New Zealand Metadata Extractor.
 * It supports service description to facilitate discovery, allows
 * parameters to be discovered and submitted to control the
 * underlying tools (if needed).
 * @author Peter Melms (peter.melms@uni-koeln.de), Andrew Jackson
 *         <Andrew.Jackson@bl.uk>
 */

@WebService(name = Characterise.NAME,
            targetNamespace = PlanetsServices.NS)
@BindingType(value =
             "http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true")
public interface Characterise extends PlanetsService {

  /** The service name. */
  String NAME = "Characterise";
  /** The qualified name. */
  QName QNAME = new QName(PlanetsServices.NS, Characterise.NAME);

  /**
   * @param digitalObject The digital object to characterise
   * @param parameters for fine grained tool control
   * @return A list of properties, wrapped into a CharacteriseResult
   */
  @WebMethod(operationName = Characterise.NAME,
             action = PlanetsServices.NS + "/" + Characterise.NAME)
  @WebResult(name = Characterise.NAME + "Result",
             targetNamespace = PlanetsServices.NS + "/" +
                 Characterise.NAME,
             partName = Characterise.NAME + "Result")
  @RequestWrapper(className = "eu.planets_project.services.characterise."
                      + Characterise.NAME + "Characterise")
  @ResponseWrapper(className =
                       "eu.planets_project.services.characterise."
                       + Characterise.NAME + "CharacteriseResponse")
```

```
  CharacteriseResult characterise(
        @WebParam(name = "digitalObject",
                  targetNamespace = PlanetsServices.NS
                       + "/" + Characterise.NAME,
                  partName = "digitalObject")
          final DigitalObject digitalObject,
        @WebParam(name = "parameters",
                  targetNamespace = PlanetsServices.NS
                        + "/" + Characterise.NAME,
                  partName = "parameters") List<Parameter> parameters);

  /**
   * @param formatURI A format URI
   * @return The properties this characterisation service extracts
   *         for the given file format
   */
  @WebMethod(operationName = Characterise.NAME + "_" + "listProperties",
             action = PlanetsServices.NS
                 + "/" + Characterise.NAME + "/" + "listProperties")
  @WebResult(name = Characterise.NAME + "Property_List",
             targetNamespace = PlanetsServices.NS
                 + "/" + Characterise.NAME,
             partName = Characterise.NAME + "Property_List")
  @ResponseWrapper(className =
                     "eu.planets_project.services.characterise."
                     + Characterise.NAME + "listPropertiesResponse")
  List<Property> listProperties(URI formatURI);
}
```

A.7.2 The CaracteriseResult Source Code

```
/**
 *
 */
package eu.planets_project.services.characterise;

import eu.planets_project.services.datatypes.Property;
import eu.planets_project.services.datatypes.ServiceReport;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;

import java.net.URI;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/**
 * Result type for the {@link Characterise} interface.
 * @author <a href="mailto:Andrew.Jackson@bl.uk">Andy Jackson</a>
 * @author <a href="mailto:fabian.steeg@uni-koeln.de">Fabian Steeg</a>
 * (recursive structure)
 */
@XmlRootElement
@XmlAccessorType(value = XmlAccessType.FIELD)
public final class CharacteriseResult {

  private ServiceReport report;

  private String fragmentID;

  private List<Property> props;

  private URI formatURI;

  /*
   * Proposed way to support embedded results: embedding
   * CharacteriseResult itself, instead of introducing a new class
   * or reusing some other class. This seems to be the most
   * straightforward approach. It keeps the simple case simple
   * and does not change the API for it.
   * Also, for the sample scenario of text files with embedded images,
   * I believe it makes sense: we have a separate characterisation
   * result for each embedded object, aggregated into a result for
   * the complete object.
   */
  private List<CharacteriseResult> results;

  /**
   * For JAXB.
   */
  @SuppressWarnings("unused")
  private CharacteriseResult() {}

  /**
   * @param props The characterisation result properties
   * @param report The service report
   */
  public CharacteriseResult(final List<Property> props,
                            final ServiceReport report) {
```

```
    this.props = props;
    this.report = report;
    this.results = new ArrayList<CharacteriseResult>();
    this.fragmentID = null;
    this.formatURI = null;
  }

  /**
   * @param props The characterisation result properties
   * @param report The service report
   * @param fragmentID The id of the fragment the results are for
   */
  public CharacteriseResult(final List<Property> props,
                            final ServiceReport report,
                            final String fragmentID) {
    this.props = props;
    this.report = report;
    this.results = new ArrayList<CharacteriseResult>();
    this.fragmentID = fragmentID;
    this.formatURI = null;
  }

  /**
   * @param props The characterisation result properties
   * @param report The service report
   * @param formatURI A format URI for the Characterise Result
   */
  public CharacteriseResult(final List<Property> props,
                            final ServiceReport report,
                            final URI formatURI) {
    this.props = props;
    this.report = report;
    this.results = new ArrayList<CharacteriseResult>();
    this.fragmentID = null;
    this.formatURI = formatURI;
  }

  /**
   * @param props The characterisation result properties
   * @param report The service report
   * @param fragmentID The id of the fragment the results are for
   * @param formatURI A format URI for the Characterise Result
   */
  public CharacteriseResult(final List<Property> props,
                            final ServiceReport report,
                            final String fragmentID,
                            final URI formatURI) {
    this.props = props;
    this.report = report;
    this.results = new ArrayList<CharacteriseResult>();
    this.fragmentID = fragmentID;
    this.formatURI = formatURI;
  }

  /**
   * @param props The characterisation result properties
   * @param report The service report
   * @param results The embedded results
   */
  public CharacteriseResult(final List<Property> props,
                            final ServiceReport report,
                            final List<CharacteriseResult> results) {
    this.props = props;
```

```
    this.report = report;
    this.results = results;
    this.fragmentID = null;
    this.formatURI = null;
  }

  /**
   * @param props The characterisation result properties
   * @param report The service report
   * @param results The embedded results
   * @param fragmentID The id of the fragment the results are for
   */
  public CharacteriseResult(final List<Property> props,
                            final ServiceReport report,
                            final List<CharacteriseResult> results,
                            final String fragmentID) {
    this.props = props;
    this.report = report;
    this.results = new ArrayList<CharacteriseResult>();
    this.fragmentID = fragmentID;
    this.formatURI = null;
  }

  /**
   * @param props The characterisation result properties
   * @param report The service report
   * @param results The embedded results
   * @param formatURI A format URI for the Characterise Result
   */
  public CharacteriseResult(final List<Property> props,
                            final ServiceReport report,
                            final List<CharacteriseResult> results,
                            final URI formatURI) {
    this.props = props;
    this.report = report;
    this.results = new ArrayList<CharacteriseResult>();
    this.fragmentID = null;
    this.formatURI =formatURI;
  }

  /**
   * @param props The characterisation result properties
   * @param report The service report
   * @param results The embedded results
   * @param fragmentID The id of the fragment the results are for
   * @param formatURI A format URI for the Characterise Result
   */
  public CharacteriseResult(final List<Property> props,
                            final ServiceReport report,
                            final List<CharacteriseResult> results,
                            final String fragmentID,
                            URI formatURI) {
    this.props = props;
    this.report = report;
    this.results = new ArrayList<CharacteriseResult>();
    this.fragmentID = fragmentID;
    this.formatURI = formatURI;
  }

  /**
   * @return An unmodifiable view of the result properties
   */
  public List<Property> getProperties() {
```

```
    // The extra check here is necessary as JAXB unmarshalls
    // empty lists to null.
    return Collections.unmodifiableList(props == null ?
        new ArrayList<Property>() : props);
  }

  /**
   * @return The service report
   */
  public ServiceReport getReport() {
    return report;
  }

  /**
   * @return An unmodifiable view of the embedded results
   */
  public List<CharacteriseResult> getResults() {
    // The extra check here is necessary as JAXB unmarshalls
    // empty lists to null.
    return Collections.unmodifiableList(results == null ?
            new ArrayList<CharacteriseResult>() : results);
  }

  /**
   * @return The fragment ID
   */
  public String getFragmentID() {
    return this.fragmentID;
  }

  /**
   * @return The format URI
   */
  public URI getFormatURI() {
    return this.formatURI;
  }
}
```

A.8 The Compare Interface

A.8.1 The Compare Interface Source Code

```
/**
 *
 */
package eu.planets_project.services.compare;

import java.util.List;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingType;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;

import eu.planets_project.services.PlanetsService;
import eu.planets_project.services.PlanetsServices;
import eu.planets_project.services.datatypes.DigitalObject;
import eu.planets_project.services.datatypes.Parameter;

/**
 * Comparison of digital objects.
 * @author Fabian Steeg (fabian.steeg@uni-koeln.de)
 */
@WebService(name = Compare.NAME, targetNamespace = PlanetsServices.NS)
@BindingType(value =
                  "http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true")
public interface Compare extends PlanetsService {
  /***/
  String NAME = "Compare";
  /***/
  QName QNAME = new QName(PlanetsServices.NS, Compare.NAME);

  /**
   * @param first The first of the two digital objects to compare
   * @param second The second of the two digital objects to compare
   * @param config A configuration parameter list
   * @return A list of result properties, the result of
   *         comparing the given digital object, wrapped in a result
   *         object
   */
  @WebMethod(operationName = Compare.NAME,
             action = PlanetsServices.NS + "/" + Compare.NAME)
  @WebResult(name = Compare.NAME + "Result",
             targetNamespace = PlanetsServices.NS + "/" + Compare.NAME,
             partName = Compare.NAME + "Result")
  @RequestWrapper(className = "eu.planets_project.services.compare."
                      + Compare.NAME + "Request")
  @ResponseWrapper(className = "eu.planets_project.services.compare."
                       + Compare.NAME + "Response")
  CompareResult compare(@WebParam(name = "first",
                                  targetNamespace = PlanetsServices.NS
                                      + "/" + Compare.NAME,
                                  partName = "firstDigitalObject")
                        final DigitalObject first,
                        @WebParam(name = "second",
                                  targetNamespace = PlanetsServices.NS
```

```
                                        + "/" + Compare.NAME,
                                    partName = "secondDigitalObject")
                            final DigitalObject second,
                            @WebParam(name = "config",
                                    targetNamespace = PlanetsServices.NS
                                        + "/" + Compare.NAME,
                                    partName = "config")
                            final List<Parameter> config);

    /**
     * Convert a tool-specific configuration file to the generic
     * format of a list of properties. Use this method to pass your
     * configuration file to {@link #compare(DigitalObject,
     *               DigitalObject, List)}.
     * @param configFile The tool-specific configuration file
     * @return A list of parameters containing the configuration values
     */
    @WebMethod(operationName = "ConfigProperties",
                action = PlanetsServices.NS + "/" + Compare.NAME)
    @WebResult(name = Compare.NAME + "ConfigProperties",
                targetNamespace = PlanetsServices.NS
                    + "/" + Compare.NAME,
                partName = Compare.NAME + "ConfigProperties")
    List<Parameter> convert(@WebParam(name = "configFile",
                            targetNamespace = PlanetsServices.NS
                                + "/" + Compare.NAME,
                            partName = "configFile")
                            final DigitalObject configFile);
}
```

A.8.2 The CompareProperties Interface

```
/**
 *
 */
package eu.planets_project.services.compare;

import java.util.List;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingType;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;

import eu.planets_project.services.PlanetsService;
import eu.planets_project.services.PlanetsServices;
import eu.planets_project.services.characterise.CharacteriseResult;
import eu.planets_project.services.datatypes.DigitalObject;
import eu.planets_project.services.datatypes.Parameter;

/**
 * Comparison of CharacteriseResult objects (the output of the
 * Characterise interface).
 * @see eu.planets_project.services.characterise.Characterise
 * @author Fabian Steeg (fabian.steeg@uni-koeln.de)
 */
@WebService(name = CompareProperties.NAME,
            targetNamespace = PlanetsServices.NS)
@BindingType(value =
                 "http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true")
public interface CompareProperties extends PlanetsService {
  /***/
  String NAME = "CompareProperties";
  /***/
  QName QNAME = new QName(PlanetsServices.NS, CompareProperties.NAME);

  /**
   * @param first The first of the two CharacteriseResult objects
   *              to compare
   * @param second The second of the two CharacteriseResult objects
   *               to compare
   * @param config A configuration parameter list
   * @return A list of result properties, the result of comparing
   *         the given digital object, wrapped in a result object
   */
  @WebMethod(operationName = CompareProperties.NAME,
             action = PlanetsServices.NS + "/" + CompareProperties.NAME)
  @WebResult(name = CompareProperties.NAME + "Result",
             targetNamespace = PlanetsServices.NS
                 + "/" + CompareProperties.NAME,
             partName = CompareProperties.NAME + "Result")
  @RequestWrapper(className = "eu.planets_project.services.compare."
                      + CompareProperties.NAME + "Request")
  @ResponseWrapper(className = "eu.planets_project.services.compare."
                       + CompareProperties.NAME + "Response")
  CompareResult compare(@WebParam(name = "first",
                                  targetNamespace = PlanetsServices.NS
                                      + "/" + CompareProperties.NAME,
```

```
                                    partName = "firstCharacteriseResult")
                        final CharacteriseResult first,
                        @WebParam(name = "second",
                                    targetNamespace = PlanetsServices.NS
                                        + "/" + CompareProperties.NAME,
                                    partName = "secondCharacteriseResult")
                        final CharacteriseResult second,
                        @WebParam(name = "config",
                                    targetNamespace = PlanetsServices.NS
                                        + "/" + CompareProperties.NAME,
                                    partName = "config")
                        final List<Parameter> config);

    /**
     * Convert a tool-specific input file (like the output of a
     * characterisation tool) to the generic format the service uses.
     * Use this method to pass as the first two arguments to {@link
     * #compare(CharacteriseResult, CharacteriseResult, List)}.
     *
     * @param inputFile The tool-specific configuration file
     * @return A CharacteriseResult object representing the
     *         given input file
     */
    @WebMethod(operationName = "InputProperties",
                action = PlanetsServices.NS + "/" +
                     CompareProperties.NAME)
    @WebResult(name = CompareProperties.NAME + "InputProperties",
                targetNamespace = PlanetsServices.NS
                    + "/" + CompareProperties.NAME,
                partName = CompareProperties.NAME + "InputProperties")
    CharacteriseResult convertInput(
              @WebParam(name = "inputProperties",
                        targetNamespace = PlanetsServices.NS
                            + "/" + CompareProperties.NAME,
                        partName = "inputProperties")
                final DigitalObject inputFile);

    /**
     * Convert a tool-specific configuration file to the generic format
     * of a list of properties. Use this method to pass your
     * configuration file as the last argument to {@link
     * #compare(CharacteriseResult, CharacteriseResult, List)}.
     * @param configFile The tool-specific configuration file
     * @return A list of parameters containing the configuration values
     */
    @WebMethod(operationName = CompareProperties.NAME + "ConfigProperties",
                action = PlanetsServices.NS + "/" + CompareProperties.NAME)
    @WebResult(name = CompareProperties.NAME + "ConfigProperties",
                targetNamespace = PlanetsServices.NS
                    + "/" + CompareProperties.NAME,
                partName = CompareProperties.NAME + "ConfigProperties")
    List<Parameter> convertConfig(
          @WebParam(name = "configFile",
                    targetNamespace = PlanetsServices.NS
                            + "/" + CompareProperties.NAME,
                    partName = "configFile")
          final DigitalObject configFile);
}
```

A.8.3 The CompareResult Source Code

```
/**
 *
 */
package eu.planets_project.services.compare;

import eu.planets_project.services.datatypes.Property;
import eu.planets_project.services.datatypes.ServiceReport;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

/**
 * Result type for the {@link Compare} interface.
 * @author Fabian Steeg (fabian.steeg@uni-koeln.de)
 */
@XmlRootElement
@XmlAccessorType(value = XmlAccessType.FIELD)
public final class CompareResult {
  private List<Property> properties;
  private ServiceReport report;
  private String fragmentID;
  private List<CompareResult> results;

  /** For JAXB. */
  @SuppressWarnings("unused")
  private CompareResult() {}

  /**
   * @param properties The result properties
   * @param report The report
   */
  public CompareResult(final List<Property> properties,
                       final ServiceReport report) {
    if( properties != null ) {
      this.properties = new ArrayList<Property>(properties);
    } else {
      this.properties = new ArrayList<Property>();
    }
    this.report = report;
    this.results =
        Collections.unmodifiableList(new ArrayList<CompareResult>());
    this.fragmentID = null;
  }

  /**
   * @param properties The result properties
   * @param report The report
   * @param fragmentID the id of the fragment the result refers to
   */
  public CompareResult(final List<Property> properties,
                       final ServiceReport report,
                       final String fragmentID) {
    this.properties = new ArrayList<Property>(properties);
    this.report = report;
    this.results =
        Collections.unmodifiableList(new ArrayList<CompareResult>());
```

```
    this.fragmentID = fragmentID;
  }

  /**
   * @param properties The result properties
   * @param report The report
   * @param results The embedded results
   */
  public CompareResult(final List<Property> properties,
                       final ServiceReport report,
                       final List<CompareResult> results) {
    this.properties = new ArrayList<Property>(properties);
    this.report = report;
    this.results = new ArrayList<CompareResult>(results);
    this.fragmentID = null;
  }

  /**
   * @param properties The result properties
   * @param report The report
   * @param results The embedded results
   * @param fragmentID the id of the fragment the result refers to
   */
  public CompareResult(final List<Property> properties,
                       final ServiceReport report,
                       final List<CompareResult> results,
                       final String fragmentID) {
    this.properties = new ArrayList<Property>(properties);
    this.report = report;
    this.results = new ArrayList<CompareResult>(results);
    this.fragmentID = fragmentID;
  }
  /**
   * @return An unmodifiable copy of the result properties
   */
  public List<Property> getProperties() {
    return properties == null ?
        Collections.unmodifiableList(new ArrayList<Property>()) :
        properties;
  }

  /**
   * @return The report
   */
  public ServiceReport getReport() {
    return report;
  }

  /**
   * @return An unmodifiable copy of the embedded results
   */
  public List<CompareResult> getResults() {
    return results == null ?
        Collections.unmodifiableList(new ArrayList<CompareResult>()) :
        results;
  }

  /**
   * @return The fragment ID
   */
  public String getFragmentID() {
    return this.fragmentID;
  }
```

```
}
```

A.9 The Migrate Interface

A.9.1 The Migrate Interface Source Code

```
/**
 *
 */
package eu.planets_project.services.migrate;

import java.net.URI;
import java.util.List;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingType;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;
import javax.xml.ws.soap.SOAPBinding;

import eu.planets_project.services.PlanetsService;
import eu.planets_project.services.PlanetsServices;
import eu.planets_project.services.datatypes.DigitalObject;
import eu.planets_project.services.datatypes.Parameter;

/**
 * Migration of one digital object.
 *
 * This is intended to become the generic migration interface for
 * complex migration services.
 *
 * It should:
 *  - Support service description to facilitate discovery.
 *  - Allow multiple input formats and output formats to be
 *    dealt with be the same service.
 *  - Allow parameters to be discovered and submitted to control
 *    the migration.
 *  - Allow digital objects composed of more than one file/bitstream.
 *  - Allow Files/bitstreams passed by value OR by reference.
 *
 * @author Fabian Steeg (fabian.steeg@uni-koeln.de),
 * Andrew Jackson <Andrew.Jackson@bl.uk>
 */
@WebService(name = Migrate.NAME,
            targetNamespace = PlanetsServices.NS)
@BindingType(value = SOAPBinding.SOAP11HTTP_MTOM_BINDING )
public interface Migrate extends PlanetsService {
  /** The interface name */
  String NAME = "Migrate";
  /** The qualified name */
  QName QNAME = new QName(PlanetsServices.NS, Migrate.NAME);

  /**
   * Migrate one digital object from inputFormat to outputFormat.
   *
   *
   * Note: The migration should ignore the formatURI specified
   * in the digital object.
   *
   * @param digitalObject The digital object to migrate
```

```
    * @param inputFormat the initial format (migrate from)
    * @param outputFormat the required format (migrate to)
    * @param parameters a list of parameters to provide fine
    *        grained tool control
    * @return A new digital object, the result of migrating
    *         the given digital
    *         object
    */
   @WebMethod(operationName = Migrate.NAME,
              action = PlanetsServices.NS + "/" + Migrate.NAME)
   @WebResult(name = Migrate.NAME + "Result",
              targetNamespace = PlanetsServices.NS + "/" + Migrate.NAME,
              partName = Migrate.NAME + "Result")
   @RequestWrapper(className="eu.planets_project.services.migrate." +
                       Migrate.NAME+"Migrate")
   @ResponseWrapper(className="eu.planets_project.services.migrate." +
                        Migrate.NAME+"MigrateResponse")
   public MigrateResult migrate(
         @WebParam(name = "digitalObject",
                   targetNamespace = PlanetsServices.NS
                       + "/" + Migrate.NAME,
                   partName = "digitalObject")
         final DigitalObject digitalObject,
         @WebParam(name = "inputFormat",
                   targetNamespace = PlanetsServices.NS
                         + "/" + Migrate.NAME,
                   partName = "inputFormat")
         URI inputFormat,
         @WebParam(name = "outputFormat",
                   targetNamespace = PlanetsServices.NS
                         + "/" + Migrate.NAME,
                   partName = "outputFormat")
         URI outputFormat,
         @WebParam(name = "parameters",
                   targetNamespace = PlanetsServices.NS
                       + "/" + Migrate.NAME,
                   partName = "parameters")
         List<Parameter> parameters );
}
```

A.9.2 The MigrateResult Source Code

```
/**
 *
 */
package eu.planets_project.services.migrate;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;

import eu.planets_project.services.datatypes.DigitalObject;
import eu.planets_project.services.datatypes.ServiceReport;

/**
 * Result type for migration services.
 * @author <a href="mailto:Andrew.Jackson@bl.uk">Andy Jackson</a>
 */
@XmlRootElement
@XmlAccessorType(value = XmlAccessType.FIELD)
public final class MigrateResult {

  private DigitalObject digitalObject;
  private ServiceReport report;

  /**
   * For JAXB.
   */
  @SuppressWarnings("unused")
  private MigrateResult() {}

  /**
   * @param digitalObject The resulting digital object
   * @param report The report for this migration
   */
  public MigrateResult(final DigitalObject digitalObject,
                       final ServiceReport report) {
    this.digitalObject = digitalObject;
    this.report = report;
  }

  /**
   * @return the digitalObject
   */
  public DigitalObject getDigitalObject() {
    return digitalObject;
  }

  /**
   * @return the event
   */
  public ServiceReport getReport() {
    return report;
  }
}
```

A.10 The Modify Interface

A.10.1 The Modify Interface Source Code

```
/**
 *
 */
package eu.planets_project.services.modify;

import java.net.URI;
import java.util.List;

import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.WebService;
import javax.xml.namespace.QName;
import javax.xml.ws.BindingType;
import javax.xml.ws.RequestWrapper;
import javax.xml.ws.ResponseWrapper;

import eu.planets_project.services.PlanetsService;
import eu.planets_project.services.PlanetsServices;
import eu.planets_project.services.datatypes.DigitalObject;
import eu.planets_project.services.datatypes.Parameter;

/**
 * Interface for services modifying digital objects.
 * @author Peter Melms
 *
 */
@WebService(name = Modify.NAME,
            targetNamespace = PlanetsServices.NS)
@BindingType(value =
                "http://schemas.xmlsoap.org/wsdl/soap/http?mtom=true")
public interface Modify extends PlanetsService {
  /** The interface name */
  String NAME = "Modify";
  /** The qualified name */
  QName QNAME = new QName(PlanetsServices.NS, Modify.NAME);

  /**
   * Modify a given object.
   * @param digitalObject the digital object
   * @param inputFormat The input format
   *        (this will probably be removed in a subsequent release)
   * @param parameters The parameters, if any
   * @return A modify result response object
   */
  @WebMethod(operationName = Modify.NAME,
             action = PlanetsServices.NS + "/" + Modify.NAME)
  @WebResult(name = Modify.NAME + "Result",
             targetNamespace = PlanetsServices.NS + "/" + Modify.NAME,
             partName = Modify.NAME + "Result")
  @RequestWrapper(className="eu.planets_project.services.modify." +
                      Modify.NAME + "Modify")
  @ResponseWrapper(className="eu.planets_project.services.modify." +
                       Modify.NAME + "ModifyResponse")
  public ModifyResult modify(
    @WebParam(name = "digitalObject",
              targetNamespace = PlanetsServices.NS + "/" + Modify.NAME,
              partName = "digitalObject")
```

```
    final DigitalObject digitalObject,
    @WebParam(name = "inputFormat",
              targetNamespace = PlanetsServices.NS + "/" + Modify.NAME,
              partName = "inputFormat")
    final URI inputFormat,
    @WebParam(name = "parameters",
              targetNamespace = PlanetsServices.NS + "/" + Modify.NAME,
              partName = "parameters")
    List<Parameter> parameters );
}
```

A.10.2 The ModifyResult Source Code

```
/**
 *
 */
package eu.planets_project.services.modify;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;

import eu.planets_project.services.datatypes.DigitalObject;
import eu.planets_project.services.datatypes.ServiceReport;

/**
 * Result type for modification services.
 * @author <a href="mailto:Peter.Melms@uni-koeln.de">Peter Melms</a>
 */
@XmlRootElement
@XmlAccessorType(value = XmlAccessType.FIELD)
public final class ModifyResult {

  private DigitalObject digitalObject;
  private ServiceReport report;

  /**
   * for JAXB.
   */
  @SuppressWarnings("unused")
  private ModifyResult() {};

  /**
   * @param digitalObject The modified digital object
   * @param report The report for the modification
   */
  public ModifyResult(final DigitalObject digitalObject,
                      final ServiceReport report) {
    this.digitalObject = digitalObject;
    this.report = report;
  }

  /**
   * @return The digital object
   */
  public DigitalObject getDigitalObject() {
    return digitalObject;
  }

  /**
   * @return The report
   */
  public ServiceReport getReport() {
    return report;
  }
}
```

Appendix B Service Datatypes

B.1 Introduction

This Appendix gives more information about the datatypes that support the IF Service Interfaces. The content is a copy of the project Wiki documentation and the source code with comments.

B.2 Digital Object

B.2.1 Introduction

This section provides:

- A brief technical overview of the digital object, clarifies the purpose of the fields within the class and gives some guidelines as to their use.
- A brief guide to using DigitalObjects.
- A guide to other resources available
- A snapshot of the DigitalObject code.

B.2.2 Technical Overview

The DigitalObject consists of two classes:

- DigitalObject (interface)
- ImmutableDigitalObject (implemenation)

While the fields discussed below are really from the ImmutableDigitalObject, for reasonable purposes they can be regarded as fields of the DigitalObject.

B.2.3 The Digital Object Fields

The aim of the section is not to provide detailed technical documentation, which can be found in the code comments / JavaDoc. It presents a brief description of the intended use of the fields within the class and examples of the information that they should or shouldn't contain. Each heading has two names, one is the name of the Java class member, the other is the name of the element found in the XML representation. They are shown as **JavaName <XmlName>**, as the two aren't always identical.

title <title>

String, **Required**

A String title for the digital object, this is the name which identifies the digital object. It doesn't need to be universally unique, but a name like "object" will make the object hard to identify in future. Required as each object should have a human readable title.

format <format>

URI, **Optional**

A URI indicating the format of the digital object, in whatever scheme the creator knows this format. This format is regarded as not-trusted, by the planets framework, so it is not required, but should be filled in, when known.

permanentUrl <permanentUrl>

URL, **Optional**

A URL that uniquely identifies the Digital Object, it may well be the URL at which the object is located.

manifestationOf <manifestationOf>

URI, **Optional**

The URI that this digital object is a manifestation of. This digital object contain data, and this data is a digital manifistation of some work. The URI should be the URI of this work.

metadata <metadata>

List of eu.planets_project.services.datatypes.Metadata, **Optional**

These hold additional, repository-specific metadata.

See IF/Services/Datatypes/Metadata

contained <contained>

List of DigitalObject, **Optional**

If this DigitalObject is a folder, or some other structure holding other digital objects in a readily accesible way, they should be stored in the contained field.

content <content>

eu.planets_project.services.datatypes.Content, **Required**

An object that holds the binary content or a reference to it.

See IF/Services/Datatypes/Content

events <events>

List of eu.planets_project.services.datatypes.Event, **Optional**

This is a complete record of all of the Planets events for this DigitalObject, effectively a Digital Preservation audit trail, history. Objects does not need to have such a audit trail when made, but whenever an action is taken on the object, the audit trail should be updated.

See IF/Services/Datatypes/Event

fragments <fragments>

List of eu.planets_project.services.datatypes.Fragment, **Optional**

This list is meant to hold the names of the files in a compound object, such as a zip file. There are no usage examples for this field as yet, it is anticipated that it will be required for more complex DigitalObjects the documentation will be updated in time.

B.2.4 How to Use a DigitalObject

A Digital Object has:

Metadata

- URL permanentUrl;
- String title;
- URI planetsFormatUri;
- URI manifestationOf;
- List<Metadata> taggedMetadata;
- <Event> events;

Children

- List<DigitalObject> contained;

Content (Bytestream)

- List<Content> content; (Content is by reference (URL) or by value)
- Checksum checksum;

Bytestream Structures

- <Fragment> fragments;

Digital Object Examples

A Digital Object will always need an accurate Format URI in order to be interpreted unambiguously. This is particularly true for multi-component objects. In the short term, Planets may have to develop it's own identifiers.

A Single File

A single file wrapped as a digital object must have

- A single Content bytestream.
- NO child objects (contained == null).

This type should probably also have a PRONOM ID for the format URI to identify the format of the bytestream.

A Single File with Structure

If an Single File and its associate bytestream has some structure that is of interest, then the Fragment parts will be used to describe this structure. Fragment types will probably have to be defined specifically for different Digital Object types, and are essentially a special case of Digital Object Characteristics.

A Composite Object

A Digital Object composed of multiple files must have

- Two or more child Digital Objects.

In this case, to resolve the contents, the recipient must descend into the children and determine their content.

The layout of the children will depend on the digital object type and so require a format URI. For example, we may wish to define a 'sequence of images' composite object (planets:fmt/comp/seq/image, say) and define this as a simple container digital object that wraps a flat list of child objects, with a sequence dictated by the order in which they appear in the list.

Further details of these complex content types should be driven by use cases.

A Packaged Composite Object

A self-contained composite Digital Object must have:

- Two or more child Digital Objects.
- A top level Content object that contains or points to the entity that contains the bytestreams referenced by the child Digital Objects.

The nature of the entity that contains the bytestreams will depend on the digital object type and so require a suitable format URI. For a ZIP archive containing arbitrary files, the Content would be or point to the ZIP file bytestream. For a simple directory/folder, the Content would point to the folder, perhaps using a file URL.

Further details of these complex content types should be driven by use cases.

Storage of Digital Objects

Retrieving a Digital Object from a Repository

When a Digital Object is pulled from a Repository, it must have

- A permanentUrl that uniquely identifies this entity in the context of the handling workflow. Generally, this will mean a globally unique web URL under some scheme like HTTP. In the special case of a workflow that knows it is operating locally, this may be a simple file:// URL.

Saving a Digital Object into a Repository

When a Digital Object is put into a repository, the repository must decide what information should be retained. In most cases, it should also assign a new permanentUrl for this new item.

Digital Objects in IF Services

In general, it is up to the caller (workflow) to decide whether to pass a Digital Object by value or reference, and in the latter case to use a scheme that the callee should be able to understand. e.g. If the workflow knows the services are local, it should be able to use a file:// URL. In the case where the service is remote, the simplest method is to embed the binary.

For service implementations, the developers should simply fail and report if they were unable to access the URL. If the service creates a new Digital Object, e.g. in Migration, it is up to the service implementer to decide whether to embed the binaries in the return call or to create new references and embed those. The migration interface provides a suggested write location, probably a temporary directory, and in this case the service can write the suggested location and does not need to worry about managing these files. Before writing, the service should check that the write location exists, is accessible, and is empty, and refuse to write otherwise.

B.2.5 The DigitalObject Source Code

```
package eu.planets_project.services.datatypes;

import javax.xml.bind.annotation.adapters.XmlAdapter;
import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;
import java.net.URI;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

/**
 * A representation of a digital object. Instances are created using a
 * builder
 * to allow optional named constructor parameters and ensure consistent
 * state during creation. E.g. to create a digital object with only
 * the required argument, you'd use:
 * <p/>
 * {@code DigitalObject o = new DigitalObject.Builder(content).build();}
 * <p/>
 * You can cascade additional calls for optional arguments:
 * <p/>
 * {@code DigitalObject o = new
 * DigitalObject.Builder(content).manifestationOf(abstraction).
 *   title(title).build();}
 * <p/>
 * DigitalObject instances can be serialized to XML. Given such an XML
 * representation, a digital object can be instantiated using the
 * builder:
 * <p/>
 * {@code DigitalObject o = new DigitalObject.Builder(xml).build();}
 * <p/>
 * For usage examples, see the tests in {@link DigitalObjectTests} and
 * web service sample usage in pserv/IF/simple.
 * @author Fabian Steeg
 */
@XmlJavaTypeAdapter(DigitalObject.Adapter.class)
public interface DigitalObject {

  /** @return The title of this digital object. */
  String getTitle();

  /** @return The type of this digital object. */
  URI getFormat();

  /** @return The unique identifier. */
  URI getPermanentUri();

  /** @return The URI that this digital object is a manifestation of. */
  URI getManifestationOf();

  /**
   * @return Additional repository-specific metadata. Returns a defensive
   *     copy, changes to the obtained list won't affect this digital
   *     object.
```

```
   */
  List<Metadata> getMetadata();

  /**
   * @return The actual content reference. Required.
   *     Returns a defensive copy, changes to the obtained instance
   *     won't affect this digital object.
   */
    DigitalObjectContent getContent();

  /**
   * @return The 0..n events that happened to this digital object.
   *      Returns a defensive copy, changes to the obtained list
   *      won't affect this digital object.
   */
  List<Event> getEvents();

  /**
   * @return The 0..n fragment IDs this digital object consists of.
   *     Returns a defensive copy, changes to the obtained list
   *         won't affect this digital object. If required, a future
   *         version of the framework  might use a complex type to
   *         represent a fragment.
  */
  List<String> getFragments();

  /**
   * @return An XML representation of this digital object
   *         (can be used to instantiate an object using the builder
   *          constructor)
  */
  String toXml();

  /* Same approach as above, but for the DigitalObject itself. */
  /** Adapter for serialization of DigitalObject interface instances. */
  static class Adapter extends
        XmlAdapter<ImmutableDigitalObject, DigitalObject> {
    /**
     * {@inheritDoc}
     * @see
javax.xml.bind.annotation.adapters.XmlAdapter#unmarshal(java.lang.Object)
    */
    public DigitalObject unmarshal(final ImmutableDigitalObject o) {
        return o;
    }

    /**
     * {@inheritDoc}
     * @see
javax.xml.bind.annotation.adapters.XmlAdapter#marshal(java.lang.Object)
     */
    public ImmutableDigitalObject marshal(final DigitalObject o) {
        return (ImmutableDigitalObject) o;
    }
  }

  /**
   * Builder for DigitalObject instances. Using a builder ensures
   *    consistent object state during creation and models optional
   *    named constructor parameters.
   * @see eu.planets_project.services.datatypes.DigitalObjectTests
   */
  public static final class Builder {
```

```
    /* Required parameter: */
    private DigitalObjectContent content;
    /* Optional parameters, initialized to default values: */
    private URI permanentUri = null;
    private List<Event> events = new ArrayList<Event>();
    private List<String> fragments = new ArrayList<String>();
    private URI manifestationOf = null;
    private List<Metadata> metadata = new ArrayList<Metadata>();
    private URI format = null;
    private String title = null;

    /** @return The instance created using this builder. */
    public DigitalObject build() {
      return new ImmutableDigitalObject(this);
    }

    /**
     * Constructs an anonymous (permanentUri == null) digital object.
     * To set further attributes, call the desired methods on the
     * resulting builder.
     * @param content The content of the digital object,
     *                see static factory methods in {@link Content}
     *                for different ways of content creation,
     *                e.g. {@code Content.byReference(file)}.
     */
    public Builder(final DigitalObjectContent content) {
      this.content = content;
    }

    /**
     * @param digitalObject An existing digital object to copy into a new
     *        anonymous (permanentUri == null) digital object.
     */
    public Builder(final DigitalObject digitalObject) {
        content = digitalObject.getContent();
        events = digitalObject.getEvents();
        fragments = digitalObject.getFragments();
        manifestationOf = digitalObject.getManifestationOf();
        title = digitalObject.getTitle();
        metadata = digitalObject.getMetadata();
        format = digitalObject.getFormat();
    }

    /**
     * Creates a builder that will build a digital object identical
     * to the given object, including the permanent URI.
     * @param digitalObjectXml An XML representation of a digital object.
     */
    public Builder(final String digitalObjectXml) {
        if (digitalObjectXml == null) {
          throw new IllegalArgumentException("Cannot create digital
object for null string");
        }
    /*
     * Besides the adapter, this is the second place where we mention
     * the implementation class, but as before, this is behind the
     * interface.
     */
    ImmutableDigitalObject digitalObject = ImmutableDigitalObject
            .of(digitalObjectXml);
      permanentUri = digitalObject.getPermanentUri();
      content = digitalObject.getContent();
      events = digitalObject.getEvents();
```

```
      fragments = digitalObject.getFragments();
      manifestationOf = digitalObject.getManifestationOf();
      title = digitalObject.getTitle();
      metadata = digitalObject.getMetadata();
      format = digitalObject.getFormat();
    }

    /** No-arg constructor for JAXB. */
    @SuppressWarnings("unused")
    private Builder() {}

    /**
     * @param content The new content for the digital object to be
     * created, see static factory methods in {@link Content} for
     * different ways of content creation,
     * e.g. {@code Content.byReference(file)}.
     * @return The builder, for cascaded calls
     */
    public Builder content(final DigitalObjectContent content) {
      this.content = content;
      return this;
    }

    /**
     * @param permanentUri The globally unique identifier for this
     * digital object.
     * @return The builder, for cascaded calls
     */
    public Builder permanentUri(final URI permanentUri) {
      this.permanentUri = permanentUri;
      return this;
    }

    /**
     * @param events The events of the digital object
     * @return The builder, for cascaded calls
     */
    public Builder events(final Event... events) {
      this.events = new ArrayList<Event>(Arrays.asList(events));
      return this;
    }

    /**
     * @param fragments The fragments the digital object is made of
     * @return The builder, for cascaded calls
     */
    public Builder fragments(final String... fragments) {
      this.fragments = new ArrayList<String>(Arrays.asList(fragments));
      return this;
    }

    /**
     * @param manifestationOf
     *     What the digital object is a manifestation of
     * @return The builder, for cascaded calls
     */
    public Builder manifestationOf(final URI manifestationOf) {
      this.manifestationOf = manifestationOf;
      return this;
    }

    /**
     * @param title The title of the digital object
```

```
     * @return The builder, for cascaded calls
     */
    public Builder title(final String title) {
      this.title = title;
      return this;
    }

    /**
     * @param metadata Additional metadata for the digital object
     * @return The builder, for cascaded calls
     */
    public Builder metadata(final Metadata... metadata) {
      this.metadata = new ArrayList<Metadata>(Arrays.asList(metadata));
      return this;
    }

    /**
     * @param format The type of the digital object
     * @return The builder, for cascaded calls
     */
    public Builder format(final URI format) {
      this.format = format;
      return this;
    }

    /**
     * @return The content
     * @see DigitalObject#getContent()
     */
    public DigitalObjectContent getContent() {
      return content;
    }

    /**
     * @return The permanent URI
     * @see DigitalObject#getPermanentUri()
     */
    public URI getPermanentUri() {
      return permanentUri;
    }

    /**
     * @return The events
     * @see DigitalObject#getEvents()
     */
    public List<Event> getEvents() {
      return events;
    }

    /**
     * @return The fragments
     * @see DigitalObject#getFragments()
     */
    public List<String> getFragments() {
      return fragments;
    }

    /**
     * @return The abstraction this object is a manifestation of
     * @see DigitalObject#getManifestationOf()
     */
    public URI getManifestationOf() {
      return manifestationOf;
```

```
    }

    /**
     * @return The metadata
     * @see DigitalObject#getMetadata()
    */
    public List<Metadata> getMetadata() {
      return metadata;
    }

    /**
     * @return The format
     * @see DigitalObject#getFormat()
    */
    public URI getFormat() {
      return format;
    }

    /**
     * @return The title
     * @see DigitalObject#getTitle()
     */
    public String getTitle() {
      return title;
    }
  }
}
```

B.2.6 The ImmutableDigitalObject Source Code

```
package eu.planets_project.services.datatypes;

import eu.planets_project.services.PlanetsServices;

import javax.xml.bind.*;
import javax.xml.bind.annotation.*;
import javax.xml.transform.Result;
import javax.xml.transform.stream.StreamResult;
import java.io.IOException;
import java.io.InputStream;
import java.io.Serializable;
import java.io.StringReader;
import java.io.StringWriter;
import java.net.URI;
import java.util.ArrayList;
import java.util.List;

/**
 * Representation of an immutable, comparable concrete digital object,
 * to be passed through web services and serializable with JAXB.
 * As the other planets data types, it uses XmlAccessType.FIELD
 * instead of getters and setters.
 * This allows for proper encapsulation on the API side while
 * remaining JAXB-serializable.
 * <p/>
 * This class is immutable in practice; its instances can therefore
 * be shared freely and concurrently.
 * <p/>
 * A corresponding XML schema can be generated from this class by
 * running this class as a Java application, see
 * {@link #main(String[])}.
 * @author <a href="mailto:fabian.steeg@uni-koeln.de">Fabian Steeg</a>
 * @see DigitalObjectTests
 */
@XmlRootElement(name = "digitalObject", namespace =
PlanetsServices.OBJECTS_NS)
@XmlType(namespace = PlanetsServices.OBJECTS_NS)
@XmlAccessorType(value = XmlAccessType.FIELD)
/*
 * NOTE: This class is intentionally NOT PUBLIC. Clients should use
 * a DigitalObject.Builder to instantiate digitalobjects.
 */
final class ImmutableDigitalObject implements
    DigitalObject, Serializable {

  /** Generated UID. */
  private static final long serialVersionUID = -893249048201058999L;

  /** @see {@link #getTitle()} */
  @XmlAttribute(required = true)
  private String title;

  /** @see {@link #getFormat()} */
  @XmlAttribute
  private URI format;

  /** @see {@link #getPermanentUri()} */
  @XmlAttribute
  private URI permanentUri;
```

```
  /** @see {@link #getManifestationOf()} */
  @XmlAttribute
  private URI manifestationOf;

  /** @see {@link #getMetadata()} */
  @XmlElement(namespace = PlanetsServices.OBJECTS_NS)
  private List<Metadata> metadata = new ArrayList<Metadata>();

  /** @see {@link #getContent()} */
  @XmlElement(namespace = PlanetsServices.OBJECTS_NS, required = true)
  private ImmutableContent content;

  /** @see {@link #getEvents()} */
  @XmlElement(namespace = PlanetsServices.OBJECTS_NS)
  private List<Event> events = new ArrayList<Event>();

  /** @see {@link #getFragments()} */
  @XmlElement(namespace = PlanetsServices.OBJECTS_NS)
  private List<String> fragments = new ArrayList<String>();

  /**
   * @param builder The builder to construct a digital object from
   */
  ImmutableDigitalObject(final Builder builder) {
    permanentUri = builder.getPermanentUri();
    /*
     * FIXME: We cast here to allow using the concrete content
     * implementation class in this digital object implementation
     * in order to avoid having to make the content interface
     * serializable (which this digital object implementation is,
     * but not digital objects in general).
     * While this is safe in our current setup, it is not
     * ideal, because this digital object implementation can't be
     * combined with a different content implementation.
     * We currently need Java Serialization support since GUI
     *  components we use require it.
     */
    content = (ImmutableContent) builder.getContent();
    events = builder.getEvents();
    fragments = builder.getFragments();
    manifestationOf = builder.getManifestationOf();
    title = builder.getTitle();
    metadata = builder.getMetadata();
    format = builder.getFormat();
  }

  /**
   * No-args constructor for JAXB serialization.
   *  Should not be called by an API client. Clients should use:
   * <p/>
   * {@code new DigitalObject.Builder(required args...)
   *    optional args...build();}
   */
  @SuppressWarnings("unused")
  private ImmutableDigitalObject() {}

  /**
   * @param xml The XML representation of a digital object
   * (as created from calling toXml)
   * @return A digital object instance created from the given XML
   */
  static ImmutableDigitalObject of(final String xml) {
    try {
```

```
      /* Unmarshall with JAXB: */
      JAXBContext context =
          JAXBContext.newInstance(ImmutableDigitalObject.class);
      Unmarshaller unmarshaller = context.createUnmarshaller();
      Object object = unmarshaller.unmarshal(new StringReader(xml));
      ImmutableDigitalObject unmarshalled =
         (ImmutableDigitalObject) object;
      return unmarshalled;
    } catch (JAXBException e) {
      e.printStackTrace();
    }
    return null;
  }

  /**
   * {@inheritDoc}
   * @see eu.planets_project.services.datatypes.DigitalObject#toXml()
   */
  public String toXml() {
    if (this.content.getDataHandler() != null) {
      /*
       * If the content uses a DataHandler (which is annotated with
       * @XmlAttachmentRef for streaming support),
       * normal JAXB serialization won't work
       * (see test in DataHandlerAttachmentSerialization). To get XML
       * serialization for these cases, we create a copy of the
       * digital object and create the same content, but using
       * Content.byValue:
       */
      InputStream inputStream = getContent().getInputStream();
      if (inputStream == null) {
        throw new IllegalStateException("Could not read content from: "
            + getContent());
      }
      return new
DigitalObject.Builder(this).content(Content.byValue(inputStream)).permane
ntUri(permanentUri).build().toXml();
    }
    try {
      /* Marshall with JAXB: */
      JAXBContext context =
          JAXBContext.newInstance(ImmutableDigitalObject.class);
      Marshaller marshaller = context.createMarshaller();
      StringWriter writer = new StringWriter();
      marshaller.marshal(this, writer);
      return writer.toString();
    } catch (JAXBException e) {
      e.printStackTrace();
    }
    return null;
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#toString()
   */
  public String toString() {
    int contentSize = content == null ? 0 : 1;
    String checksum = content == null || content.getChecksum() ==
        null ? "" : content.getChecksum().toString();
    int eventsSize = events == null ? 0 : events.size();
    int fragmentsSize = fragments == null ? 0 : fragments.size();
    int metaSize = metadata == null ? 0 : metadata.size();
```

```
    return String.format("DigitalObject: id '%s', title '%s'; %s content
elements, " + "%s events, %s fragments; "
       + "type '%s', manifestation of '%s', checksum '%s', metadata
       '%s'", permanentUri, title, contentSize,
       eventsSize, fragmentsSize, format, manifestationOf,
       checksum, metaSize);
  }

  /**
   * {@inheritDoc}
   * @see eu.planets_project.services.datatypes.DigitalObject#getTitle()
   */
  public String getTitle() {
      return title;
  }

  /**
   * {@inheritDoc}
   * @see eu.planets_project.services.datatypes.DigitalObject#getFormat()
   */
  public URI getFormat() {
    return format;
  }

  /**
   * {@inheritDoc}
   * @see
eu.planets_project.services.datatypes.DigitalObject#getPermanentUri()
   */
  public URI getPermanentUri() {
    return permanentUri;
  }

  /**
   * {@inheritDoc}
   * @see
eu.planets_project.services.datatypes.DigitalObject#getManifestationOf()
   */
  public URI getManifestationOf() {
    return manifestationOf;
  }

  /**
   * {@inheritDoc}
   * @see
eu.planets_project.services.datatypes.DigitalObject#getMetadata()
   */
  public List<Metadata> getMetadata() {
    return metadata == null ? null : new ArrayList<Metadata>(metadata);
  }

  /**
   * {@inheritDoc}
   * @see
eu.planets_project.services.datatypes.DigitalObject#getContent()
   */
  public DigitalObjectContent getContent() {
    return content;
  }

  /**
   * {@inheritDoc}
   * @see eu.planets_project.services.datatypes.DigitalObject#getEvents()
```

```
   */
  public List<Event> getEvents() {
    return events == null ? null : new ArrayList<Event>(events);
  }

  /**
   * {@inheritDoc}
   * @see
eu.planets_project.services.datatypes.DigitalObject#getFragments()
   */
  public List<String> getFragments() {
    return fragments == null ? null : new ArrayList<String>(fragments);
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#hashCode()
   */
  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((content == null) ?
        0 : content.hashCode());
    result = prime * result + ((events == null) ? 0 : events.hashCode());
    result = prime * result + ((format == null) ? 0 : format.hashCode());
    result = prime * result + ((fragments == null) ?
        0 : fragments.hashCode());
    result = prime * result + ((manifestationOf == null) ?
        0 : manifestationOf.hashCode());
    result = prime * result + ((metadata == null) ?
        0 : metadata.hashCode());
    result = prime * result + ((permanentUri == null) ?
        0 : permanentUri.hashCode());
    result = prime * result + ((title == null) ?
        0 : title.hashCode());
    return result;
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#equals(java.lang.Object)
   */
  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    ImmutableDigitalObject other = (ImmutableDigitalObject) obj;
    if (content == null) {
      if (other.content != null)
        return false;
     } else if (!content.equals(other.content))
        return false;
    if (events == null) {
      if (other.events != null)
        return false;
    } else if (!events.equals(other.events))
      return false;
    if (format == null) {
```

```
      if (other.format != null)
        return false;
    } else if (!format.equals(other.format))
      return false;
    if (fragments == null) {
      if (other.fragments != null)
        return false;
    } else if (!fragments.equals(other.fragments))
      return false;
    if (manifestationOf == null) {
      if (other.manifestationOf != null)
        return false;
    } else if (!manifestationOf.equals(other.manifestationOf))
      return false;
    if (metadata == null) {
      if (other.metadata != null)
        return false;
    } else if (!metadata.equals(other.metadata))
      return false;
    if (permanentUri == null) {
      if (other.permanentUri != null)
        return false;
    } else if (!permanentUri.equals(other.permanentUri))
      return false;
    if (title == null) {
      if (other.title != null)
        return false;
    } else if (!title.equals(other.title))
      return false;
    return true;
  }

  /* Schema generation: */

  /***/
  private static java.io.File baseDir = new
       java.io.File("IF/common/src/resources");
  /***/
  private static String schemaFileName = "digital_object.xsd";

  /** Resolver for schema generation. */
  static class Resolver extends SchemaOutputResolver {
    /**
     * {@inheritDoc}
     * @see
javax.xml.bind.SchemaOutputResolver#createOutput(java.lang.String,
java.lang.String)
     */
    public Result createOutput(final String namespaceUri,
        final String suggestedFileName) throws IOException {
          return new StreamResult(new java.io.File(baseDir,
schemaFileName.split("\\.")[0] + "_" + suggestedFileName));

    }
  }

  /**
   * Generates the XML schema for this class.
   * @param args Ignored
   */
  public static void main(final String[] args) {
    try {
      Class<ImmutableDigitalObject> clazz = ImmutableDigitalObject.class;
```

```
      JAXBContext context = JAXBContext.newInstance(clazz);
      context.generateSchema(new Resolver());
      System.out.println("Generated XML schema for " +
          clazz.getSimpleName() + " at "
          + new java.io.File(baseDir, schemaFileName).getAbsolutePath());
    } catch (JAXBException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

B.3 Content

B.3.1 Introduction

This section provides:

- A brief technical overview of the digital object Content class, clarifies the purpose of the various fields within the class and gives some guidance as to their use.
- A snapshot of the source code.

B.3.2 Technical Overview

Content is in fact two classes:

- Content (interface)
- ImmutableContent (the implementation)

Whereas the fields discussed below are really fields of ImmutableContent class, they can for all reasonable purposes be regarded as fields of Content.

B.3.3 The Content Fields

The aim of the section is not to provide detailed technical documentation that can be found in the code comments / JavaDoc. It presents a brief description of the intended use of the fields within the class and examples of the information they should and shouldn't contain.

Each heading has two names, one is the name of the Java class member, the other is then name of the element found in the XML representation. They are shown as JavaName <xmlname>, as the two aren't always identical.

reference <reference>

URL, **Optional**

The reference to the content. You must have either reference or dataHandler filled out, but not both.

This is the preferred way to transfer content

dataHandler <dataHandler>

Octet-Stream, **Optional**

The byte array containing the content. If the content is not backed by any server, this is the only way to transfer content. Note that this is very memory intensive, and should not be used with files above 20 MB.

length <length>

long, **Autogenerated**

The length of the content, if known. The system will autogenerate this value, if capable, and there are thus no way to set it.

checksum <checksum>

Checksum, **Optional**

If the content has a checksum, this checksum can be transferred along with the content. The checksum is stored in a Checksum object, which is so simple it does not have it's own description page

- The Checksum Fields
    - Algorithm
      String, **Required**
      The name of the algorithm used to compute the checksum. The algorithm field has intentionally been left open ended, but there are reserved names for the standard algorithms.

"MD5"
"SHA-1"
"CRC32"

- Value
  String, **Required**
  The value of the checksum. This value will only make sense in the context of the algorithm.

B.3.4 The Content Source Code

```
package eu.planets_project.services.datatypes;

import eu.planets_project.services.utils.FileUtils;

import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.net.URL;

/**
 * Static factory methods for content creation.
 * @author Fabian Steeg (fabian.steeg@uni-koeln.de)
 */
public final class Content {

  private Content() {/* enforce non-instantiability */};

  /*
   * We use static factory methods to provide named constructors for the
   * different kinds of content instances:
   */

  /**
   * Create content by reference.
   * @param reference The URL reference to the actual content
   * @return A content instance referencing the given location
   */
  public static DigitalObjectContent byReference(final URL reference) {
    return new ImmutableContent(reference);
  }

  /**
   * Create (streamed) content by reference, from a file.
   * Note that the file must be left in place long enough for the
   * web service client to complete
   * the access.
   * @param reference The reference to the actual content value,
   *                  using a File whose content will be streamed
   *                  over the connection.
   * @return A content instance referencing the given location.
   */
  public static DigitalObjectContent byReference(final File reference) {
    if (!reference.exists()) {
      throw new IllegalArgumentException("Given file does not exist: "
            + reference);
    }
    return new ImmutableContent(reference);
  }

  /**
   * Create (streamed) content by reference, from an input stream.
   * @param inputStream The InputStream containing the value
   *                    for the content.
```

```
   * @return A content instance with the specified value
   */
  public static DigitalObjectContent byReference
        (final InputStream inputStream) {
    /* Write the stream to a temp file (is guaranteed to create a
     * new file, given string is for base name only)
     */
    File tmpFile = null;
    try {
      tmpFile = File.createTempFile("tempContent", "tmp");
      tmpFile.deleteOnExit();
    } catch (IOException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
    FileUtils.writeInputStreamToFile(inputStream, tmpFile);
    return new ImmutableContent(tmpFile);
  }

  /**
   * Create content by value (actually embedded in the request).
   * @param value The value bytes for the content
   * @return A content instance with the specified value
   */
  public static DigitalObjectContent byValue(final byte[] value) {
    return new ImmutableContent(value);
  }

  /**
  * Create content by value (actually embedded in the request).
  * @param value The value file for the content.
  * @return A content instance with the specified value
  */
  public static DigitalObjectContent byValue(final File value) {
    if (!value.exists()) {
      throw new IllegalArgumentException("Given file does not exist: "
          + value);
    }
    byte[] bytes = FileUtils.readFileIntoByteArray(value);
    return new ImmutableContent(bytes);
  }

  /**
   * Create content by value (actually embedded in the request).
   * @param inputStream The InputStream containing the value for
   *                    the content.
   * @return A content instance with the specified value
   */
  public static DigitalObjectContent byValue(final InputStream
inputStream) {
    return new
ImmutableContent(FileUtils.writeInputStreamToBinary(inputStream));
  }
}
```

B.3.5 The ImmutableContent Source Code

```
package eu.planets_project.services.datatypes;

import java.io.ByteArrayInputStream;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.io.Serializable;
import java.net.URL;
import java.util.Arrays;
import java.util.logging.Logger;

import javax.activation.DataHandler;
import javax.activation.FileDataSource;
import javax.activation.FileTypeMap;
import javax.xml.bind.annotation.XmlAttachmentRef;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlMimeType;
import javax.xml.bind.annotation.XmlType;

import org.apache.commons.io.IOUtils;

import com.sun.xml.ws.developer.StreamingDataHandler;

import eu.planets_project.services.PlanetsServices;

/**
 * Content for digital objects, either by reference or by value.
 * @see ContentTests
 * @author Asger Blekinge-Rasmussen (abr@statsbiblioteket.dk)
 * @author Fabian Steeg (fabian.steeg@uni-koeln.de)
 * @author Peter Melms (peter.melms@uni-koeln.de)
 */
@XmlType(namespace = PlanetsServices.OBJECTS_NS)
/*
 * NOTE: This class is intentionally NOT PUBLIC. Clients should use
 * the factory methods in the Content class to instantiate content.
 */
final class ImmutableContent implements
        DigitalObjectContent, Serializable {
  private static Logger log =
          Logger.getLogger(ImmutableContent.class.getName());

  private static final long serialVersionUID = 7135127983024589335L;

  /***/
  @XmlAttribute
  private URL reference;

  @XmlElement
  private byte[] bytes;

  @XmlElement(namespace = PlanetsServices.OBJECTS_NS)
  @XmlMimeType("application/octet-stream")
  /*
   * FIXME: This field is non-serializable and non-transient.
   * We can't make it serializable because it's not ours and
   * we can't make it transient because then JAXB complains.
   * Support for Java Serialization is currently required by
   * GUI components. Possible solutions: using different UI
```

```
 * components; using some sort of wrapper object in the GUI
 * (SerializableDigitalObject).
 */
@XmlAttachmentRef() //This appears to be required to actually enable
                    //the streaming data handler
private DataHandler dataHandler;

/***/
@XmlAttribute
private long length = -1;

@XmlElement(namespace = PlanetsServices.OBJECTS_NS)
private Checksum checksum = null;

/**
 * @param value The content value
 */
ImmutableContent(final byte[] value) {
  if (value == null)
    throw new IllegalArgumentException("Byte array parameter must” +
                                       “not be null!");
    this.length = value.length;
    this.bytes = value;
    log.info("Created Content from byte array: " + this.length +
             " bytes in length.");
}

/**
 * @param reference The content reference, as a file.
 */
ImmutableContent(final File reference) {
  if (reference == null) throw new IllegalArgumentException("File” +
                                    “parameter must not be null!");
  FileDataSource ds = new FileDataSource(reference);
  ds.setFileTypeMap(FileTypeMap.getDefaultFileTypeMap());
  DataHandler dh = new DataHandler(ds);
  this.length = reference.length();
  this.dataHandler = dh;
  log.info("Created Content from file: " +
           reference.getAbsolutePath() +
           "': " + this.length + " bytes in length.");
}

/**
 * @param reference The content, passed as an explicit reference.
 */
ImmutableContent(final URL reference) {
  if (reference == null) throw new IllegalArgumentException(
                     "URL parameter must not be null!");
  this.length = -1;
  this.reference = reference;
  log.info("Created Content from URL: " + reference);
}

/** No-args constructor for JAXB. Clients should not use this. */
@SuppressWarnings("unused")
private ImmutableContent() {}

/**
 * @param immutableContent The content to copy
 * @param checksum The checksum to attach to the content copy
 */
```

```
  private ImmutableContent(final ImmutableContent immutableContent, final
Checksum checksum) {
    this.dataHandler = immutableContent.dataHandler;
    this.length = immutableContent.length;
    this.reference = immutableContent.reference;
    this.checksum = checksum;
  }

  /**
   * {@inheritDoc}
   * @see
eu.planets_project.services.datatypes.DigitalObjectContent#getInputStream
()
   */
  public InputStream getInputStream() {
    try {
      if (dataHandler != null) {
        log.info("Opening dataHandler stream of type: " +
                 dataHandler.getContentType());
        log.info("Opening dataHandler stream available: " +
                 dataHandler.getInputStream().available());
        if (dataHandler instanceof StreamingDataHandler) {
          StreamingDataHandler h = (StreamingDataHandler) dataHandler;
          return h.getInputStream(); //readOnce basically works
                                     //but makes usage inconvenient
        }
        return dataHandler.getInputStream();
      } else if (bytes != null) {
        return new ByteArrayInputStream(bytes);
      } else {
        log.info("Opening reference: " + reference);
        return reference.openStream();
      }
    } catch (IOException e) {
      e.printStackTrace();
      return null;
    }
  }

  /**
   * @return The reference, if any (might be null).
   *        Clients should not se this method to access the actual
   *        data, but {@link #getInputStream()} or {@link #getValue()},
   *        which will always return the actual content, no matter
   *        how it was created (by value or by reference).
   */
  public URL getReference() {
    return reference;
  }

  /**
   * @return The data handler, or null (if this content uses a
   *         URL or byte[])
   */
  DataHandler getDataHandler() {
    return dataHandler;
  }

  /**
   * @return True, if this Content contains the actual value,
   *               or false if it contains a reference
   */
  public boolean isByValue() {
```

```
    return reference == null && dataHandler == null;
  }

  /**
   * {@inheritDoc}
   * @see
eu.planets_project.services.datatypes.DigitalObjectContent#length()
   */
  public long length() {
    return length;
  }

  /**
   * {@inheritDoc}
   * @see
eu.planets_project.services.datatypes.DigitalObjectContent#withChecksum(e
u.planets_project.services.datatypes.Checksum)
   */
  public DigitalObjectContent withChecksum(final Checksum checksum) {
    return new ImmutableContent(this, checksum);
  }

  /**
   * {@inheritDoc}
   * @see
eu.planets_project.services.datatypes.DigitalObjectContent#getChecksum()
   */
  public Checksum getChecksum() {
    return checksum;
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#equals(java.lang.Object)
   */
  @Override
  public boolean equals(final Object obj) {
    if (!(obj instanceof ImmutableContent)) {
      return false;
    }
    ImmutableContent that = (ImmutableContent) obj;
    /*
     * Two content objects, even if they would be based on the same
     * data, are not equal if they are not both by
     * reference or both by value:
     */
    if (this.isByValue() != that.isByValue()) {
      return false;
    }
    /* Else we compare either value or reference: */
    try {
      if (this.dataHandler != null && that.dataHandler != null) {
        return IOUtils.contentEquals(dataHandler.getInputStream(),
                  that.dataHandler.getInputStream());
      } else if (this.bytes != null && that.bytes != null) {
        return IOUtils
       .contentEquals(new ByteArrayInputStream(this.bytes),
                      new ByteArrayInputStream(that.bytes));
      } else if (this.reference != null && that.reference != null) {
        Return
          this.reference.toString().equals(that.reference.toString());
      }
    } catch (IOException e) {
```

```
      e.printStackTrace();
    }
    return false;
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#hashCode()
   */
  @Override
  public int hashCode() {
    return dataHandler != null ?
       dataHandler.hashCode() : bytes != null ?
           Arrays.hashCode(bytes) : reference.toString().hashCode();
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#toString()
   */
  @Override
  public String toString() {
    return String.format("Content by %s: %s", isByValue() ?
        "value" : "reference", dataHandler != null ?
            dataHandler : bytes != null ?
                Arrays.asList(bytes) : reference);
  }
}
```

B.4 Checksum

B.4.1 Introduction

This section provides:

- A brief technical overview of the digital object Content class, clarifies the purpose of the various fields within the class and gives some guidance as to their use.
- A snapshot of the source code.

B.4.2 Technical Overview

B.4.3 The Checksum Fields

B.4.4 The ChecksumSource Code

```
package eu.planets_project.services.datatypes;

import eu.planets_project.services.PlanetsServices;

import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlType;
import java.io.Serializable;

/**
 * Immutable representation of a checksum, containing of the algorithm
used and
 * the actual value.
 * @see ChecksumTests
 * @author Fabian Steeg
 */
@XmlType(namespace = PlanetsServices.OBJECTS_NS)
public final class Checksum implements Comparable<Checksum>, Serializable
{
  /** Generated UID. */
```

```
  private static final long serialVersionUID = 8799717233710485566L;
  /** @see #getAlgorithm() */
  @XmlAttribute
  private String algorithm; /* Not final for JAXB */
  /** @see #getValue() */
  @XmlAttribute
  private String value;

  /**
   * @param algorithm The checksum algorithm.
   * @param value The checksum value.
   */
  public Checksum(final String algorithm, final String value) {
    this.algorithm = algorithm;
    this.value = value;
  }

  /** No-args constructor for JAXB usage. */
  @SuppressWarnings("unused")
  private Checksum() {}

  /**
   * @return The checksum algorithm.
   */
  public String getAlgorithm() {
    return algorithm;
  }

  /**
   * @return The checksum value.
   */
  public String getValue() {
    return value;
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#equals(java.lang.Object)
   */
  @Override
  public boolean equals(final Object obj) {
    if (!(obj instanceof Checksum)) {
      return false;
    }
    Checksum other = (Checksum) obj;
    return this.compareTo(other) == 0;
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#toString()
   */
  @Override
  public String toString() {
    return String.format("[%s, algorithm: %s, value: %s]",
                         this.getClass().getSimpleName(),
                         algorithm, value);
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Comparable#compareTo(java.lang.Object)
   */
```

```
  public int compareTo(final Checksum o) {
    if (this.algorithm.equals(o.algorithm)) {
      return this.value.compareTo(o.value);
    }
    return this.algorithm.compareTo(o.algorithm);
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#hashCode()
   */
  @Override
  public int hashCode() {
    /* Following 'Effective Java'... */
    int i = 17;
    int j = 31;
    int result = i;
    result = j * result + algorithm.hashCode();
    result = j * result + value.hashCode();
    return result;
  }
}
```

B.5 Metadata

B.5.1 Introduction

This section provides:

- A brief technical overview of the digital object Content class, clarifies the purpose of the various fields within the class and gives some guidance as to their use.
- A snapshot of the source code.

B.5.2 Technical Overview

B.5.3 The Metadata Fields

B.5.4 The Metadata Source Code

```
package eu.planets_project.services.datatypes;

import eu.planets_project.services.PlanetsServices;

import javax.xml.bind.annotation.*;
import java.io.Serializable;
import java.net.URI;

/**
 * Representation of immutable tagged metadata.
 * @see MetadataTests
 * @author Fabian Steeg (fabian.steeg@uni-koeln.de)
 */
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(namespace = PlanetsServices.OBJECTS_NS)
public final class Metadata implements Comparable<Metadata>, Serializable
{
  /** Generated UID. */
  private static final long serialVersionUID = 1299020544765389245L;

  /**
   * The block of metadata.
   * @see #getContent()
   */
  @XmlElement(namespace = PlanetsServices.OBJECTS_NS, required = true)
```

```
  private String content;

  /**
   * @see Metadata#getType()
   */
  @XmlAttribute(required = true)
  private URI type;

  /**
   * @param type The metadata type. Represents the type of metadata.
   *             The URI could be the namespace of a xml schema,
   *             or a xml datatype like integer. But, in short,
   *             given the URI, you should be able to figure out how
   *             to understand the metadata. No URI means that the
   *             metadata is readily readable, ie. clear text.
   * @param content The actual metadata
   */
  public Metadata(final URI type, final String content) {
    this.type = type;
    this.content = content;
  }

  /** No-args constructor for JAXB. */
  @SuppressWarnings("unused")
  private Metadata() {}

  /**
   * @return The actual metadata.
   */
  public String getContent() {
    return content;
  }

  /**
   * @return The metadata type.
   */
  public URI getType() {
    return type;
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#equals(java.lang.Object)
   */
  @Override
  public boolean equals(final Object obj) {
    return obj instanceof Metadata &&
            this.compareTo((Metadata) obj) == 0;
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#hashCode()
   */
  @Override
  public int hashCode() {
    /* Following 'Effective Java'... */
    int i = 17;
    int j = 31;
    int result = i;
    result = j * result + type.hashCode();
    result = j * result + content.hashCode();
    return result;
```

```
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#toString()
   */
  @Override
  public String toString() {
    return String.format("Metadata of type '%s' with content: %s",
                         type, content);
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Comparable#compareTo(java.lang.Object)
   */
  public int compareTo(final Metadata other) {
    if (this.type.equals(other.type)) {
      return this.content.compareTo(other.content);
    }
    return this.type.compareTo(other.type);
  }
}
```

B.6 MigrationPath

B.6.1 Introduction

This section provides:

- A brief technical overview of the MigrationPath class, clarifies the purpose of the various fields within the class and gives some guidance as to their use.
- A snapshot of the source code.

B.6.2 Technical Overview

B.6.3 The MigrationPath Fields

B.6.4 The MigrationPath Source Code

```
/**
 *
 */
package eu.planets_project.services.datatypes;

import eu.planets_project.services.PlanetsServices;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;
import java.net.URI;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;

/**
 * Simple class to build path matrices from. Contains the input and
 * outputs of the path, and allows for parameters for that mapping.
 *
 * @author <a href="mailto:Andrew.Jackson@bl.uk">Andy Jackson</a>
 */
@XmlType(name = "path", namespace = PlanetsServices.SERVICES_NS)
@XmlAccessorType(value = XmlAccessType.FIELD)
public final class MigrationPath {
```

```
  /**
   * A particular input format.
   */
  URI inputFormat;
  /**
   * The output format.
   */
  URI outputFormat;
  /**
   * The parameters that specifically apply to this pathway.
   */
  List<Parameter> parameters;

  /**
   * For JAXB.
   */
  @SuppressWarnings("unused")
  private MigrationPath() {}

  /**
   * Parameterised constructor.
   * @param in The input format
   * @param out The output format
   * @param pars The parameters
   */
  public MigrationPath(URI in, URI out, List<Parameter> pars) {
    this.inputFormat = in;
    this.outputFormat = out;
    this.parameters = pars;
  }

  /**
   * @return the inputFormat
   */
  public URI getInputFormat() {
    return inputFormat;
  }

  /**
   * @return the outputFormat
   */
  public URI getOutputFormat() {
    return outputFormat;
  }

  /**
   * @return A copy of the parameters
   */
  public List<Parameter> getParameters() {
    return new ArrayList<Parameter>(parameters);
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#toString()
   */
  @Override
  public String toString() {
    return inputFormat + " -> " + outputFormat + "  Parameters: "
                + parameters;
  }

  /**
```

```
   * Construct an array of migration paths, linking all the formats
   * in input formats to all the formats in output formats.
   * If either is null or empty, the array will be length 0.
   * All migrationPaths will be with null
   * parameters.
   * @param inputFormats The allowed input formats
   * @param outputFormats The allowed output formats
   * @return An array of all the paths.
   */
  public static List<MigrationPath> constructPaths(Set<URI> inputFormats,
            Set<URI> outputFormats) {
    if (inputFormats == null || outputFormats == null) {
      return new ArrayList<MigrationPath>();
    } else {
      List<MigrationPath> paths = new ArrayList<MigrationPath>(
                  inputFormats.size() * outputFormats.size());
      for (URI in : inputFormats) {
        for (URI out : outputFormats) {
          if( in != null && out != null) {
             paths.add(new MigrationPath(in, out, null));
          }
        }
      }
      return paths;
    }
  }

  /**
   * Construct an array of migrationpaths, linking all the formats in
   * inputformas to all the formats in outputformats. If either is null
   * or empty, the array will be length 0. All migrationPaths will be
   * with null parameters.
   * @param inputformats The allowed inputformats
   * @param outputFormats The allowed outputformats
   * @param params the parameters for this migration path.
   * @return An array of all the paths.
   */
  public static List<MigrationPath> constructPathsWithParams(
          Set<URI> inputformats, Set<URI> outputFormats,
          List<Parameter> params) {
    if (inputformats == null || outputFormats == null) {
      return new ArrayList<MigrationPath>();
    }

    if (params == null) {
      return constructPaths(inputformats, outputFormats);
    }

    if (params.size() > 0) {
      List<MigrationPath> paths = new ArrayList<MigrationPath>(
                    inputformats.size() * outputFormats.size());
      for (URI in : inputformats) {
        for (URI out : outputFormats) {
          paths.add(new MigrationPath(in, out, params));
        }
      }
      return paths;
    } else {
      return constructPaths(inputformats, outputFormats);
    }
  }
}
```

B.7 Parameter

B.7.1 Introduction

This section provides:

- A brief technical overview of the Parameter class, clarifies the purpose of the various fields within the class and gives some guidance as to their use.
- A snapshot of the source code.

B.7.2 Technical Overview

B.7.3 The Parameter Fields

B.7.4 The Parameter Source Code

```
/**
 *
 */
package eu.planets_project.services.datatypes;

import eu.planets_project.services.PlanetsServices;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlType;

/**
 * This wraps the concept of a service parameter. When retrieved from
 * a service, the default values should be set. This form does not
 * allow optional v. required parameters, as ALL parameters should be
 * explicitly specified. An 'optional' parameter implies an implicit
 * default that would end up not being recorded in the audit trail.
 * @author AnJackson
 */
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(namespace = PlanetsServices.SERVICES_NS)
public final class Parameter {

  private String name;
  private String value;
  private String type;
  private String description;

  /* ------------------------------------------------- */

  /**
   * For JAXB.
   */
  @SuppressWarnings("unused")
  private Parameter() {}

  /**
   * Constructor for the most common case: a Parameter with name and
   * value. For parameters with optional values (type, description),
   * use a Parameter.Builder.
   * @param name A name for the parameter. Must be uniquely meaningful
   * to the service, but is not expected to carry any meaning out
   * with the service.
   * @param value The value for this parameter. Should be set to
   * the default by the service when parameter discovery is happening.
   */
  public Parameter(final String name, final String value) {
    this.name = name;
    this.value = value;
```

```
  }

  /* ------------------------------------------------------------ */

  /**
   * @param builder The builder to create a Parameter from
   */
  private Parameter(final Builder builder) {
    this.name = builder.name;
    this.value = builder.value;
    this.type = builder.type;
    this.description = builder.description;
  }

  /**
   * @return the value
   */
  public String getValue() {
    return value;
  }

  /**
   * @return the name
   */
  public String getName() {
    return name;
  }

  /**
   * @return the type
   */
  public String getType() {
    return type;
  }

  /**
   * @return the description
   */
  public String getDescription() {
    return description;
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#toString()
   */
  public String toString() {
    return this.name + " = " + this.value;
  }

  /**
   * Builder for Parameters with optional values (type, description).
   * @author Fabian Steeg (fabian.steeg@uni-koeln.de)
   */
  public static final class Builder {
    private String name;
    private String value;
    private String type;
    private String description;

    /** For JAXB. */
    @SuppressWarnings("unused")
    private Builder() {}
```

```
    /**
     * @param name The Parameter name, see
     *        {@link Parameter#Parameter(String, String)}
     * @param value The Parameter value, see
     *        {@link Parameter#Parameter(String, String)}
     */
    public Builder(final String name, final String value) {
      this.name = name;
      this.value = value;
      this.type = "";
      this.description = "";
    }

    /**
     * @param type This is a String to hold the type, which should map to
     *     the xsd types and should be assumed to be a String if empty or
     *     null. In the future, we might add limits/validation?
     *     XSD-style?
     * @return This builder, for cascaded calls
     */
    public Builder type(final String type) {
      this.type = type;
      return this;
    }

    /**
     * @param description The description of this parameter/value pair.
     *        Might be used to give further information on the possible
     *        values and their meaning.
     * @return This builder, for cascaded calls
     */
    public Builder description(final String description) {
      this.description = description;
      return this;
    }

    /**
     * @return The built Parameter object
     */
    public Parameter build() {
      return new Parameter(this);
    }
  }
}
```

B.8 Property

B.8.1 Introduction

This section provides:

- A brief technical overview of the digital object Property class, clarifies the purpose of the various fields within the class and gives some guidance as to their use.
- A snapshot of the source code.

B.8.2 Technical Overview

B.8.3 The Property Fields

B.8.4 The Property Source Code

```
/**
 *
```

```
 */
package eu.planets_project.services.datatypes;

import eu.planets_project.services.PlanetsServices;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;
import java.net.URI;

/**
 * Property representation using an URI as ID.
 * <p>
 * For the most common case (a property with ID, name and value), use the
{@link #Property(URI, String, String)}
 * constructor:
 * </p>
 * <p>
 * {@code Property p = new Property(uri, name, value);}
 * </p>
 * Only the ID is actually required. To create properties with less or
more attributes, use a {@link Property.Builder}:
 * <p>
 * {@code Property p = new Property.Builder(uri).unit(unit).build();}
 * </p>
 * <p>
 * Instances of this class are immutable and so can be shared.
 * </p>
 * @author Andrew Jackson
 * @author Fabian Steeg
 */
@XmlType(name = "property", namespace = PlanetsServices.DATATYPES_NS)
@XmlAccessorType(XmlAccessType.FIELD)
public final class Property {

  @XmlElement(namespace = PlanetsServices.DATATYPES_NS)
  private URI uri = null;
  @XmlElement(namespace = PlanetsServices.DATATYPES_NS)
  private String name = "";
  @XmlElement(namespace = PlanetsServices.DATATYPES_NS)
  private String value = "";
  @XmlElement(namespace = PlanetsServices.DATATYPES_NS)
  private String unit = "";
  @XmlElement(namespace = PlanetsServices.DATATYPES_NS)
  private String description = "";
  @XmlElement(namespace = PlanetsServices.DATATYPES_NS)
  private String type = "";

  /** For JAXB. */
  @SuppressWarnings("unused")
  private Property() {}

  /**
   * Create a property with id, name and value.
   * To create properties with less or more attributes, use a
   * {@link Property.Builder} instead.
   * @param uri The property ID
   * @param name The property name
   * @param value The property value
   */
  public Property(final URI uri, final String name, final String value) {
    this.uri = uri;
```

```
    this.name = name;
    this.value = value;
  }

  /**
   * @param builder The builder to create a property from
   */
  private Property(final Property.Builder builder) {
    this.uri = builder.uri;
    this.name = builder.name;
    this.value = builder.value;
    this.description = builder.description;
    this.unit = builder.unit;
    this.type = builder.type;
  }

  /**
   * @return the uri
   */
  public URI getUri() {
    return uri;
  }

  /**
   * @return the name
   */
  public String getName() {
    return name;
  }

  /**
   * @return the value
   */
  public String getValue() {
    return value;
  }

  /**
   * @return the unit
   */
  public String getUnit() {
    return unit;
  }

  /**
   * @return the description
   */
  public String getDescription() {
    return description;
  }

  /**
   * @return the type
   */
  public String getType() {
    return type;
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#toString()
   */
  @Override
```

```
  public String toString() {
    return String.format("%s [%s] '%s' = '%s'
                         (description=%s unit=%s type=%s)",
                         this.getClass().getSimpleName(),
                         uri, name, value, description, unit, type);
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#hashCode()
   */
  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    /*
     * Our defaults plus the fact that we are immutable guarantee
     * that no attribute can ever be null, so we can skip
     * the tedious null checks here. FIXME: not true, only the
     * builder checks this, not the one constructor...
     */
    result = prime * result + description.hashCode();
    result = prime * result + name.hashCode();
    result = prime * result + type.hashCode();
    result = prime * result + unit.hashCode();
    result = prime * result + uri.hashCode();
    result = prime * result + value.hashCode();
    return result;
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#equals(java.lang.Object)
   */
  @Override
  public boolean equals(final Object obj) {
    if (!(obj instanceof Property)) {
      return false;
    }
    /* We know we have an instance of Property now,
     * so casting is safe.
     */
    Property that = (Property) obj;
    /*
     * Our defaults plus the fact that we are immutable guarantee
     * that no attribute can ever be null, so we can skip
     * the tedious null checks here. FIXME: not true,
     * only the builder checks this, not the one constructor...
     */
    return this.uri.equals(that.uri) &&
           this.name.equals(that.name) &&
           this.value.equals(that.value) &&
           this.type.equals(that.type) &&
           this.unit.equals(that.unit) &&
           this.description.equals(that.description);
  }

  /**
   * Builder to create property instances with optional attributes.
   * @author Fabian Steeg (fabian.steeg@uni-koeln.de)
   */
  public static final class Builder {
    /* URI is required: */
```

```
    private final URI uri;
    /* Defaults for optional values are set here: */
    private String name = "";
    private String value = "";
    private String description = "";
    private String unit = "";
    private String type = "";
    /**
     * @param uri The property id
     * @throws IllegalArgumentException if the given URI is null
     */
    public Builder(final URI uri) {
      if (uri == null) {
        throw new IllegalArgumentException("Property ID uri must” +
                                           “ not be null!");
      }
      this.uri = uri;
    }

    /**
     * @param name The property name
     * @return This builder, for cascaded calls
     */
    public Builder name(final String name) {
      this.name = name;
      return this;
    }

    /**
     * @param value The property value
     * @return This builder, for cascaded calls
     */
    public Builder value(final String value) {
      this.value = value;
      return this;
    }

    /**
     * @param description The property description
     * @return This builder, for cascaded calls
     */
    public Builder description(final String description) {
      this.description = description;
      return this;
    }

    /**
     * @param unit The property unit
     * @return This builder, for cascaded calls
     */
    public Builder unit(final String unit) {
      this.unit = unit;
      return this;
    }

    /**
     * @param type The property type
     * @return This builder, for cascaded calls
     */
    public Builder type(final String type) {
      this.type = type;
      return this;
    }
```

```
    /**
     * @return The finished immutable property instance
     */
    public Property build() {
      return new Property(this);
    }
  }

  /**
   * This is a convenience method, equivalent to
   * "new Property(ServiceDescription.PROPERTY,
   * ServiceDescription.AUTHORIZED_ROLES, roles)".
   * @param roles The authorized roles, comma-separated
   *        (e.g. "admin,provider")
   * @return A property to be used to indicate the given roles are
   *         authenticated (e.g. in a ServiceDescription)
   */
  public static Property authorizedRoles(final String roles) {
    return new Property(ServiceDescription.PROPERTY,
                        ServiceDescription.AUTHORIZED_ROLES, roles);
  }
}
```

B.9 ServiceDescription

B.9.1 Introduction

This section provides:

- A brief technical overview of the digital object Content class, clarifies the purpose of the various fields within the class and gives some guidance as to their use.
- A snapshot of the source code.

B.9.2 Technical Overview

B.9.3 The ServiceDescription Fields

B.9.4 The ServiceDesctription Source Code

```
/**
 *
 */
package eu.planets_project.services.datatypes;

import eu.planets_project.services.PlanetsServices;

import javax.xml.bind.*;
import javax.xml.bind.annotation.*;
import javax.xml.transform.Result;
import javax.xml.transform.stream.StreamResult;
import java.io.IOException;
import java.io.StringReader;
import java.io.StringWriter;
import java.net.URI;
import java.net.URL;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

/**
 * A entity to hold metadata about services. The content of this object
 * was first defined at the IF meeting in September 2008. This is
```

```
 * intended to be used primarily as an XML schema, but is defined in
 * Java to make reading/writing easier.
 * See also, DOAP: http://trac.usefulinc.com/doap
 * <p/>
 * This class is immutable in practice; its instances can therefore
 * be shared freely and concurrently. Instances are created using a
 * builder to allow optional named constructor parameters and ensure
 * consistent state during creation. E.g. to create a service
 * description with only the required arguments, you'd use:
 * <p/>
 * {@code ServiceDescription d = new ServiceDescription.Builder(name,
 * type).build();}
 * <p/>
 * You can cascade additional calls for optional arguments:
 * <p/>
 * {@code ServiceDescription d = new ServiceDescription.Builder(name,
 * type).paths(path1,path2).logo(logo).build();}
 * <p/>
 * ServiceDescription instances can be serialized to XML.
 * Given such an XML representation, a service description can be
 * instantiated using a static factory method:
 * <p/>
 * {@code ServiceDescription d = ServiceDescription.of(xml);}
 * <p/>
 * To use a given service description (either as an object or as XML)
 * as a template for your service description, you can give it to the
 * builder and add or override values:
 * <p/>
 * {@code ServiceDescription d = new
 * ServiceDescription.Builder(xml).paths(path1,
 *                                       path2).logo(logo).build();}
 * <p/>
 * A corresponding XML schema can be generated from this class by
 * running this class as a Java application, see {@link #main(String[])}.
 * <p/>
 * For usage examples, see the tests in {@link ServiceDescriptionTest}.
 * @see ServiceDescriptionTest
 * @author <a href="mailto:Andrew.Jackson@bl.uk">Andy Jackson</a>, <a
 *         href="mailto:fabian.steeg@uni-koeln.de">Fabian Steeg</a>
 */
@XmlRootElement(namespace = PlanetsServices.SERVICES_NS)
@XmlType(namespace = PlanetsServices.SERVICES_NS)
@XmlAccessorType(value = XmlAccessType.FIELD)
public final class ServiceDescription {

  /** URI to use for service description properties. */
  public static final URI PROPERTY =
                URI.create("planets:property/service_description");

  /** Key to be used for properties indicating authorized roles. */
  public static final String AUTHORIZED_ROLES = "authorized_roles";

  /**
   * A brief name by which this service is known.
   */
  @XmlElement(name = "title",
              namespace = PlanetsServices.TERMS_NS,
              required = true)
  String name;

  /**
   * The name of the concrete implementation class.
   */
```

```
  @XmlElement(namespace = PlanetsServices.SERVICES_NS,
              required = true)
  String classname;

  /**
   * The type of the service, which is the fully qualified
   * name of the service interface.
   */
  @XmlElement(namespace = PlanetsServices.SERVICES_NS,
              required = true)
  String type;

  /**
   * The endpoint of the service.
   */
  @XmlElement(namespace = PlanetsServices.SERVICES_NS)
  URL endpoint;

  /**
   * Declared Parameters: [name, type, value (default)]*n.
   */
  @XmlElement(namespace = PlanetsServices.SERVICES_NS)
  List<Parameter> parameters;

  /**
   * The link to the Tool registry.
   */
  @XmlElement(namespace = PlanetsServices.TOOLS_NS, required = true)
  Tool tool;

  /**
   * Human readable description of the service. Allow to be HTML, using a
   * <![CDATA[ <b>Hi</b> ]]>
   */
  @XmlElement(namespace = PlanetsServices.TERMS_NS)
  String description;

  /**
   * Wrapper version.
   */
  @XmlElement(namespace = PlanetsServices.SERVICES_NS)
  String version;

  /**
   * Identifier - A unique identifier for this service. "We need a unique
   * id for every service; Andrew Lindley is using a MD5 hash to
   * identify a service. This is a brilliant idea. I would say this
   * field summarizes Name of class impl service, Version of service
   * and ID of Tool (URI) or makes them unnecessary."
   */
  @XmlElement(namespace = PlanetsServices.TERMS_NS)
  String identifier;

  /**
   * Who wrote the wrapper. Preferred form would be a URI or a full
   * email address, like: "Full Name <fullname@server.com>".
   */
  @XmlElement(name = "creator",
              namespace = PlanetsServices.TERMS_NS,
              required = true)
  String author;

  /**
```

```
     * The organisation that is publishing this service endpoint.
     */
    @XmlElement(name = "publisher", namespace = PlanetsServices.TERMS_NS)
    String serviceProvider;

    // FIXME Add service status....???
    /**
     * Installation instructions. Properties to be set, or s/w
     * to be installed.
     * Allow to be HTML, using non-parsed embedding, like this: <![CDATA[
     * <b>Hi</b> ]]>. JAXB should handle this.
     */
    @XmlElement(namespace = PlanetsServices.SERVICES_NS)
    String instructions;

    /**
     * Link to further information about this service wrapper.
     */
    @XmlElement(namespace = PlanetsServices.SERVICES_NS)
    URI furtherInfo;

    /**
     * A link to a web-browsable logo for this service.
     * Used when presenting the service to the user.
     */
    @XmlElement(namespace = PlanetsServices.SERVICES_NS)
    URI logo;

    /**
     * Services may specify what types they can take as inputs.
     * [input]*n This is particularly useful for Validate and Characterise.
     * If the Service perform Migration, this field should not be used. Use
     * paths instead
     * @see #paths
     */
    @XmlElement(name = "inputFormat",
                required = false,
                namespace = PlanetsServices.SERVICES_NS)
    List<URI> inputFormats;

    /**
     * Name-value pairs for service properties.
     * For characterisation services,
     * this should list all the digital object properties
     * that the service can
     * deal with.
     */
    @XmlElement(name = "property",
                required = false,
                namespace = PlanetsServices.SERVICES_NS)
    List<Property> properties;

    /**
     * If this service performs migrations, they can be listed herein:
     * Migration
     * Matrix: [input, output]*n.
     */
    @XmlElement(name = "migrationPath",
                required = false,
                namespace = PlanetsServices.SERVICES_NS)
    List<MigrationPath> paths;

    /**
```

```
   * @param name The name of the service description to build
   * @param type The type of service description to build
   * @return The builder, to alter the created service description; call
   *         build() on the builder to create the actual service
   *         description
   */
  public static ServiceDescription.Builder create(final String name,
           final String type) {
    return new ServiceDescription.Builder(name, type);
  }

  /**
   * @param serviceDescription The service description to copy
   * @return The builder, to alter the created service description; call
   *         build() on the builder to create the actual service
   *         description
   */
  public static ServiceDescription.Builder copy(
          final ServiceDescription serviceDescription) {
    return new ServiceDescription.Builder(serviceDescription);
  }

  /**
   * @param builder The builder to construct a service description from
   */
  private ServiceDescription(final Builder builder) {
    name = builder.name;
    type = builder.type;
    endpoint = builder.endpoint;
    paths = builder.paths;
    properties = builder.properties;
    inputFormats = builder.inputFormats;
    logo = builder.logo;
    furtherInfo = builder.furtherInfo;
    instructions = builder.instructions;
    serviceProvider = builder.serviceProvider;
    author = builder.author;
    identifier = builder.identifier;
    version = builder.version;
    description = builder.description;
    tool = builder.tool;
    parameters = builder.parameters;
    classname = builder.classname;
  }

  /**
   * Builder for ServiceDescription instances. Using a builder ensures
   * consistent object state during creation and models optional named
   * constructor parameters while allowing immutable objects.
   * @see eu.planets_project.services.datatypes.ServiceDescriptionTest
   */
  public static final class Builder {
    /** No-arg constructor for JAXB. API clients should not use this. */
    @SuppressWarnings("unused")
    private Builder() {}

    /* Required parameters: */
    private String name;
    private String type;
    /* Optional parameters, initialised to default values: */
    private List<MigrationPath> paths = new ArrayList<MigrationPath>();
    private List<Property> properties = new ArrayList<Property>();
    private List<URI> inputFormats = new ArrayList<URI>();
```

```
    private URI logo = null;
    private URL endpoint = null;
    private URI furtherInfo = null;
    private String instructions = null;
    private String serviceProvider = null;
    private String author = null;
    private String identifier = null;
    private String version = null;
    private String description = null;
    private Tool tool = null;
    private List<Parameter> parameters = null;
    private String classname = null;

    /** @return The instance created using this builder. */
    public ServiceDescription build() {
      return new ServiceDescription(this);
    }

    /**
     * @param name The name
     * @param type The type
     */
    public Builder(final String name, final String type) {
      this.name = name;
      this.type = type;
    }

    /**
     * @param xml The XML of a service description to use as a
     *            template for creating a new service description
     */
    public Builder(final String xml) {
      ServiceDescription d = ServiceDescription.of(xml);
      initialize(d);
    }

    /**
     * @param serviceDescription The service description to use as a
     *        template for creating a new service description
     */
    public Builder(final ServiceDescription serviceDescription) {
      initialize(serviceDescription);
    }

    /**
     * @param serviceDescription The description to use as a template for
     *        creating a new description
     */
   private void initialize(final ServiceDescription serviceDescription) {
      if( serviceDescription == null ) return;
      name = serviceDescription.name;
      type = serviceDescription.type;
      endpoint = serviceDescription.endpoint;
      paths = serviceDescription.paths;
      properties = serviceDescription.properties;
      inputFormats = serviceDescription.inputFormats;
      logo = serviceDescription.logo;
      furtherInfo = serviceDescription.furtherInfo;
      instructions = serviceDescription.instructions;
      serviceProvider = serviceDescription.serviceProvider;
      author = serviceDescription.author;
      identifier = serviceDescription.identifier;
      version = serviceDescription.version;
```

```
      description = serviceDescription.description;
      tool = serviceDescription.tool;
      parameters = serviceDescription.parameters;
      classname = serviceDescription.classname;
    }

    /**
     * @param name The service name
     * @return The builder, for cascaded calls
     */
    public Builder name(final String name) {
      this.name = name;
      return this;
    }

    /**
     * @param type The service type, i.e. the interface implemented
     * @return The builder, for cascaded calls
     */
    public Builder type(final String type) {
      this.type = type;
      return this;
    }

    /**
     * @param endpoint The endpoint for this service
     * @return The builder, for cascaded calls
     */
    public Builder endpoint(final URL endpoint) {
      this.endpoint = endpoint;
      return this;
    }

    /**
     * @param paths The migration paths supported by the service
     * @return The builder, for cascaded calls
     */
    public Builder paths(final MigrationPath... paths) {
      this.paths = new ArrayList<MigrationPath>(Arrays.asList(paths));
      return this;
    }

    /**
     * @param properties Properties for the service
     * @return The builder, for cascaded calls
     */
    public Builder properties(final Property... properties) {
      this.properties = new
                        ArrayList<Property>(Arrays.asList(properties));
      return this;
    }

    /**
     * @param inputFormats The input formats supported by the service
     * @return The builder, for cascaded calls
     */
    public Builder inputFormats(final URI... inputFormats) {
      this.inputFormats = new
                          ArrayList<URI>(Arrays.asList(inputFormats));
      return this;
    }

    /**
```

```
     * @param logo The logo
     * @return The builder, for cascaded calls
     */
    public Builder logo(final URI logo) {
      this.logo = logo;
      return this;
    }

    /**
     * @param furtherInfo Further infor on the service
     * @return The builder, for cascaded calls
     */
    public Builder furtherInfo(final URI furtherInfo) {
      this.furtherInfo = furtherInfo;
      return this;
    }

    /**
     * @param instructions The service instructions
     * @return The builder, for cascaded calls
     */
    public Builder instructions(final String instructions) {
      this.instructions = instructions;
      return this;
    }

    /**
     * @param serviceProvider The providing organization
     * @return The builder, for cascaded calls
     */
    public Builder serviceProvider(final String serviceProvider) {
      this.serviceProvider = serviceProvider;
      return this;
    }

    /**
     * @param author The service author
     * @return The builder, for cascaded calls
     */
    public Builder author(final String author) {
      this.author = author;
      return this;
    }

    /**
     * @param identifier An identifier for the service
     * @return The builder, for cascaded calls
     */
    public Builder identifier(final String identifier) {
      this.identifier = identifier;
      return this;
    }

    /**
     * @param version The service version
     * @return The builder, for cascaded calls
     */
    public Builder version(final String version) {
      this.version = version;
      return this;
    }

    /**
```

```
     * @param description A description of the service
     * @return The builder, for cascaded calls
     */
    public Builder description(final String description) {
      this.description = description;
      return this;
    }

    /**
     * @param classname The name of the class implementing the service
     * @return The builder, for cascaded calls
     */
    public Builder classname(final String classname) {
      this.classname = classname;
      return this;
    }

    /**
     * @param parameters The service parameters
     * @return The builder, for cascaded calls
     */
    public Builder parameters(final List<Parameter> parameters) {
      this.parameters = parameters;
      return this;
    }

    /**
     * @param tool The tool the service uses
     * @return The builder, for cascaded calls
     */
    public Builder tool(final Tool tool) {
      this.tool = tool;
      return this;
    }
  }
  /** For JAXB. */
  private ServiceDescription() {
    super();
  }

  /**
   * @return the name
   */
  @Queryable
  public String getName() {
    return name;
  }

  /**
   * @return the classname
   */
  @Queryable
  public String getClassname() {
    return classname;
  }

  /**
   * @return the type
   */
  @Queryable
  public String getType() {
    return type;
  }
```

```
  /**
   * @return the endpoint
   */
  @Queryable
  public URL getEndpoint() {
    return endpoint;
  }

  /**
   * @return a copy of the parameters
   */
  @Queryable
  public List<Parameter> getParameters() {
    return parameters;
  }

  /**
   * @return the tool
   */
  @Queryable
  public Tool getTool() {
    return tool;
  }

  /**
   * @return the description
   */
  @Queryable
  public String getDescription() {
    return description;
  }

  /**
   * @return the version
   */
  @Queryable
  public String getVersion() {
    return version;
  }

  /**
   * @return the author
   */
  @Queryable
  public String getAuthor() {
    return author;
  }

  /**
   * @return the serviceProvider
   */
  @Queryable
  public String getServiceProvider() {
    return serviceProvider;
  }

  /**
   * @return the instructions
   */
  @Queryable
  public String getInstructions() {
    return instructions;
```

```
  }

  /**
   * @return the furtherInfo
   */
  @Queryable
  public URI getFurtherInfo() {
      return furtherInfo;
  }

  /**
   * @return the logo
   */
  @Queryable
  public URI getLogo() {
      return logo;
  }

  /**
   * @return the identifier
   */
  @Queryable
  public String getIdentifier() {
    return identifier;
  }

  /**
   * @return the paths (unmodifiable)
   */
  @Queryable
  public List<MigrationPath> getPaths() {
    return paths == null ? new ArrayList<MigrationPath>() :
                           Collections.unmodifiableList(paths);
  }

  /**
   * @return the inputFormats (unmodifiable)
   */
  @Queryable
  public List<URI> getInputFormats() {
    return inputFormats == null ? new ArrayList<URI>() : Collections
                .unmodifiableList(inputFormats);
  }

  /**
   * @return the properties (unmodifiable)
  */
  @Queryable
  public List<Property> getProperties() {
    return properties == null ? new ArrayList<Property>() : Collections
          .unmodifiableList(properties);
  }

  /*
   * Proposed hashing and equality methods, giving the identifier
   * preference if it exists, else using the serialized XML form.
   */

  /**
   * {@inheritDoc}
   * @see java.lang.Object#hashCode()
   */
  @Override
```

```
  public int hashCode() {
    if (identifier != null) {
      return identifier.hashCode();
    } else {
       return toXml().hashCode();
        }
    }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#equals(java.lang.Object)
   */
  @Override
  public boolean equals(final Object obj) {
    if (!(obj instanceof ServiceDescription)) {
      return false;
    }
    ServiceDescription other = (ServiceDescription) obj;
    if (identifier != null) {
      return identifier.equals(other.identifier);
    } else {
      return toXml().equals(other.toXml());
    }
  }

  /**
   * @param xml The XML representation of a service description
   *            (as created from calling toXml)
   * @return A digital object instance created from the given XML
   */
  public static ServiceDescription of(final String xml) {
    try {
      /* Unmarshall with JAXB: */
      JAXBContext context = JAXBContext
                    .newInstance(ServiceDescription.class);
      Unmarshaller unmarshaller = context.createUnmarshaller();
      Object object = unmarshaller.unmarshal(new StringReader(xml));
      ServiceDescription unmarshalled = (ServiceDescription) object;
      return unmarshalled;
    } catch (JAXBException e) {
      e.printStackTrace();
    }
     return null;
  }

  /**
   * @return An XML representation of this service description
   *         (can be used to instantiate an object using the static
   *         factory method)
   */
  public String toXml() {
    return toXml(false);
  }

  /**
   * @return A formatted (pretty-printed) XML representation
   *          of this service description
   */
  public String toXmlFormatted() {
    return toXml(true);
  }

  private String toXml(boolean formatted) {
```

```
    try {
      /* Marshall with JAXB: */
      JAXBContext context = JAXBContext
              .newInstance(ServiceDescription.class);
      Marshaller marshaller = context.createMarshaller();
      StringWriter writer = new StringWriter();
      marshaller.setProperty("jaxb.formatted.output", formatted);
      marshaller.marshal(this, writer);
      return writer.toString();
    } catch (JAXBException e) {
      e.printStackTrace();
    }
    return null;
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#toString()
   */
  @Override
  public String toString() {
    return this.name + " : " + this.type + " : " + this.getDescription();
  }

  /***/
  private static java.io.File baseDir =
                    new java.io.File("IF/common/src/resources");
  /***/
  private static String schemaFileName = "service_description.xsd";

  /** Resolver for schema generation. */
  static class Resolver extends SchemaOutputResolver {
    /**
     * {@inheritDoc}
     * @see
     * javax.xml.bind.SchemaOutputResolver#createOutput(java.lang.String,
     *      java.lang.String)
     */
    public Result createOutput(final String namespaceUri,
            final String suggestedFileName) throws IOException {
      return new StreamResult(new java.io.File(baseDir, schemaFileName
              .split("\\.")[0]
              + "_" + suggestedFileName));
    }
  }

  /**
   * Generates the XML schema for this class.
   * @param args Ignored
   */
  public static void main(final String[] args) {
    try {
      Class<ServiceDescription> clazz = ServiceDescription.class;
      JAXBContext context = JAXBContext.newInstance(clazz);
      context.generateSchema(new Resolver());
      System.out.println("Generated XML schema for "
              + clazz.getSimpleName()
              + " at "
              + new java.io.File(baseDir, schemaFileName)
                      .getAbsolutePath());
    } catch (JAXBException e) {
      e.printStackTrace();
    } catch (IOException e) {
```

```
      e.printStackTrace();
    }
  }
}
```

B.10 ServiceReport

B.10.1 Introduction

This section provides:

- A brief technical overview of the digital object Content class, clarifies the purpose of the various fields within the class and gives some guidance as to their use.
- A snapshot of the source code.

B.10.2 Technical Overview

B.10.3 The ServiceReport Fields

B.10.4 The ServiceReport Source Code

```
/**
 *
 */
package eu.planets_project.services.datatypes;

import eu.planets_project.services.PlanetsServices;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;

/**
 * A report from a preservation service defined based on the need
 * identified inthe 1st and 4th Service Developers Meetings.
 * Where possible, information concerning the quality of the outputs
 * should be placed in Events associated with DigitalObjects.
 * @author <a href="mailto:Andrew.Jackson@bl.uk">Andy Jackson</a>
 * @author <a href="mailto:fabian.steeg@uni-koeln.de">Fabian Steeg</a>
 */
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(namespace = PlanetsServices.SERVICES_NS)
public final class ServiceReport {

  @XmlElement(namespace = PlanetsServices.SERVICES_NS, required = true)
  private String message;

  @XmlElement(namespace = PlanetsServices.SERVICES_NS, required = true)
  private Status status;

  @XmlElement(namespace = PlanetsServices.SERVICES_NS, required = true)
  private Type type;

  /**
   * Type of information returned by a service.
   * @author Fabian Steeg (fabian.steeg@uni-koeln.de)
   */
  @XmlType(name="ServiceReportType",
           namespace = PlanetsServices.SERVICES_NS)
  public static enum Type {
    /** Roughly corresponding to Standard Out. */
    INFO,
    /**
```

```
     * Things the user should be aware of, but are not fatal and do not
     * imply significant data loss.
     */
    WARN,
    /**
     * Serious problems invoking the service, implying that no valid
     * output will exist and the workflow should not continue.
     */
    ERROR
  }

  /**
   * Service report status.
   * @author Fabian Steeg (fabian.steeg@uni-koeln.de)
   */
  @XmlType(name="ServiceReportStatus",
           namespace = PlanetsServices.SERVICES_NS)
  public static enum Status {
    /**
     * The service was invoked successfully. This does not guarantee that
     * the service did what it was supposed to do, just that the service
     * wrapping registered no errors. For further detail,
     * examine the info string and the warn string
     */
    SUCCESS,
    /**
     * The service failed. For further details, examine the error string
     */
    TOOL_ERROR,
    /**
     * The service failed in such a way, that further invocations of this
     * service will also likely fail. Do not invoke the service
     * again<br/>
     * This is the correct error state, if the service has unfulfilled
     * dependencies from the environment, or suffers a catastrophic error
     * like OutOfMemory For further detail on the error,
     * examine the error string
     */
    INSTALLATION_ERROR
  }

  /**
   * For JAXB.
   */
  @SuppressWarnings("unused")
  private ServiceReport() {}

  /**
   * Create a service report with specified status, type and message.
   * @param type The Type enum element
   * @param status The Status enum element
   * @param message The message
   */
  public ServiceReport(final Type type, final Status status,
          final String message) {
    this.type = type;
    this.status = status;
    this.message = message;
  }

  /**
   * @return The message
   */
```

```
  public String getMessage() {
    return message;
  }

  /**
   * @return The status, an element of {@link ServiceReport.Status}
   */
  public Status getStatus() {
    return status;
  }

  /**
   * @return The type, an element of {@link ServiceReport.Type}
   */
  public Type getType() {
    return type;
  }

  /**
   * {@inheritDoc}
   * @see java.lang.Object#toString()
   */
  @Override
  public String toString() {
    return String.format(
            "ServiceReport of type '%s', status '%s', message: %s", type,
            status, message);
  }
}
```

B.11 Tool

B.11.1 Introduction

This section provides:

- A brief technical overview of the digital object Content class, clarifies the purpose of the various fields within the class and gives some guidance as to their use.
- A snapshot of the source code.

B.11.2 Technical Overview

B.11.3 The Tool Fields

B.11.4 The Tool Source Code

```
/**
 * Copyright (c) 2007, 2008 The Planets Project Partners. All rights
reserved.
 * This program and the accompanying materials are made available under
the
 * terms of the GNU Lesser General Public License v3 which accompanies
this
 * distribution, and is available at
http://www.gnu.org/licenses/lgpl.html
 */
package eu.planets_project.services.datatypes;

import eu.planets_project.services.PlanetsServices;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlType;
```

```
import java.net.MalformedURLException;
import java.net.URI;
import java.net.URL;

/**
 * A richer tool description, so that the system can work in the
 * absence of a tool registry.
 * @author <a href="mailto:Andrew.Jackson@bl.uk">Andy Jackson</a>
 */
@XmlType(namespace = PlanetsServices.TOOLS_NS)
@XmlAccessorType(value = XmlAccessType.FIELD)
public final class Tool {

  /** An identifier for this tool,
   * should be resolvable via a tool registry.
   */
  @XmlElement(namespace = PlanetsServices.TOOLS_NS)
  URI identifier;

  /** The tool name. */
  @XmlElement(namespace = PlanetsServices.TOOLS_NS)
  String name;

  /** The tool version. */
  @XmlElement(namespace = PlanetsServices.TOOLS_NS)
  String version;

  /** A tool description. */
  @XmlElement(namespace = PlanetsServices.TOOLS_NS)
  String description;

  /** A link to the tool homepage. */
  @XmlElement(namespace = PlanetsServices.TOOLS_NS)
  URL homepage;

  /** For JAXB. */
  @SuppressWarnings("unused")
  private Tool() {}

  /* ------------------------------------------ */

  /**
   * @param identifier An identifier that resolves this tool via a tool
   *        registry. Set to NULL if not in a registry.
   * @param name The name of this tool.
   * @param version The version number of this tool. NULL if unknown.
   * @param description A tool description
   * @param homepage A link to the tool homepage.
   */
  public Tool(URI identifier, String name, String version,
          String description, URL homepage) {
    this.identifier = identifier;
    this.name = name;
    this.version = version;
    this.description = description;
    this.homepage = homepage;
  }

  /* ------------------------------------------ */

  /**
   * @return the description
   */
```

```
  public String getDescription() {
    return description;
  }

  /**
   * @return the homepage
   */
  public URL getHomepage() {
    return homepage;
  }

  /**
   * @return the identifier
   */
  public URI getIdentifier() {
    return identifier;
  }

  /**
   * @return the name
   */
  public String getName() {
      return name;
  }

  /**
   * @return the version
   */
  public String getVersion() {
    return version;
  }

  /* ------------------------------------------- */

  /**
   * If you specify a tool registry URI, then any information
   * stored in the registry will be used to fill in the gaps.
   * If there is no registry URI, then you should specify at least
   * a name and version. All params can be null if unknown.
   * @param identifier An identifier resolvable via a tool registry.
   * @param name The tool name.
   * @param version The tool version.
   * @param description An optional description of the tool.
   * @param homepageUrl An optional pointer to the homepage of the tool.
   * @return A Tool instance for the given values
   */
  public static Tool create(URI identifier, String name, String version,
          String description, String homepageUrl) {
    URL homepage = null;
    try {
      homepage = new URL(homepageUrl);
    } catch (MalformedURLException e) {
      e.printStackTrace();
    }
    return new Tool(identifier, name, version, description, homepage);
  }
}
```

Appendix C Service Interface History

C.1 Service Interface History

The original set of IF service interfaces were designed using simple xsd data types such as string, byte[] and URIs. While this was effective in allowing people to produce an initial set of services, the approach was quickly found to be limiting, especially for handling binary data and allowing services to return an appropriate level of information.

A fundamental change in approach was initiated during the IF/7 Kick Off meeting at the AIT offices at the end of September 2008. Instead of using only xsd datatypes new interfaces, based around a set of IF Service Types were designed. These datatypes are simple 'struct' style classes with members that are either simple xsd types or lists of simple xsd types.

The official IF policy is that the original set of service interfaces are deprecated and not supported. All services implementing the deprecated interfaces have been altered to use the new service interface definitions.

Anyone intending to wrap a digital preservation tool as an IF service should be careful to choose a new service interface type. In order to provide clarity the service interfaces that **SHOULD NOT** be used are listed below.

C.1.1 Deprecated / Unsupported IF Interfaces

Characterisation Interfaces

```
eu.planets_project.services.characterise.BasicCharacteriseOneBinary
eu.planets_project.services.characterise.BasicCharacteriseOneBinaryXCELtoBinary
eu.planets_project.services.characterise.BasicCharacteriseOneBinaryXCELtoURI
eu.planets_project.services.identify.BasicIdentifyOneBinary
eu.planets_project.services.validate.BasicValidateOneBinary
```

Comparison Interfaces

```
eu.planets_project.services.compare.BasicCompareFormatProperties
eu.planets_project.services.compare.BasicCompareTwoXcdlReferences
eu.planets_project.services.compare.BasicCompareTwoXcdlValues
```

Format Migration Interface

```
eu.planets_project.services.migrate.BasicMigrateOneBinary
```

No further documentation for these interfaces is provided in this guide as new service development doesn't require it. The guide concentrates on providing information about the new service interfaces and the supporting datatypes.