Project Number	IST-2006-033789
Project Title	Planets
Title of Deliverable	Consolidated Release and Documentation
Deliverable Number	IF-D11
Contributing Sub-project and Work-package	IF/2, IF/4, IF/5, IF/8
Deliverable Dissemination Level	External
Deliverable Nature	Report
Contractual Delivery Date	31th March 2010
Actual Delivery Date	15th May 2010
Author(s)	Andrew Jackson, Andrew Lindley, Fabian Steeg

Abstract

This document describes the consolidated release of the Planets Interoperability Framework (IF) and should serve as a Reference Manual for that work. The purpose of this document is to provide an overview of the technical architecture (section 1), common API (section 2), digital object model (section 3), and workflow engine (section 4) of the Planets IF. For detailed documentation and working code consult the Javadoc API documentation and the unit tests, both available from the Planets GForge site:

http://gforge.planets-project.eu/gf/

Within a few weeks of the project closure (May 2010), this information will also be available as part of a SourceForge project:

http://planets-suite.sourceforge.net/

Section 1 (technical architecture) provides an overview of the aims of the Planets IF, the variations and commonalities of the partner institutions involved and the resulting consequences for the general architecture of the Planets IF.

Section 2 (common API) is organized around two central aspects of the Planets IF API: services and data. In each of these subsections, related concepts and API elements are covered, in particular the central interfaces and data types, the service registry, and the format registry.

Section 3 (digital object model) describes the central noun in the IF, the *digital object,* including its underlying schema, content representation, and serialization mechanism, as well as its storage in a data registry.

Section 4 (workflow engine) describes how to incorporate Planets Services in complex workflow operations, taking into account domain specific business logic, and how to configure, interpret, and execute these models in a controlled environment.

Keyword list

Interoperability, Architecture, Preservation Action, Framework, Web Services, Workflows

Contributors

Person	Role	Partner	Contribution
Fabian Steeg	Developer	University of Cologne (UzK)	Initial draft, common API (section 2) and digital object model (section 3)
Andrew Jackson	Developer	British Library (BL)	Section 1 – technical architecture
Andrew Lindley	Developer	Austrian Institute of Technology (AIT)	Section 4 – workflow engine

TABLE OF CONTENTS

1. Technical Architecture

1.1 Overview and Aims

Over the last four years, the Planets project has successfully implemented a service-oriented framework for digital preservation. Planets is a software suite that aims to provide the necessary infrastructure to support this, along with user applications to exploit this infrastructure for preservation purposes. From the outset, the project plan recognised the need for a clearly defined and modular software infrastructure, in order to make creation and maintenance of such a complex system manageable.

This is in common with other work, and is most closely related to the Micro Services of Abrams. See 'Preservation is not a place' http://ijdc.net/index.php/ijdc/article/viewFile/98/73, related http://www.cdlib.org/services/uc3/curation/, Merritt: An Emergent Approach to Digital Curation Infrastructure. Rev. 0.5 (2009-11-10), and Abrams' publication from iPres 2009.

In Planets, the very wide variation of the needs and requirements of the different partners have been explored and have framed the design and evolution of the software system. By analysing the variation in these requirements across partners, we were able to tease out a framework that embodies the commonality between them. This is not intended to be the final word on preservation architectures, but rather a flexible framework that allows extension and further standardisation over time as appropriate.

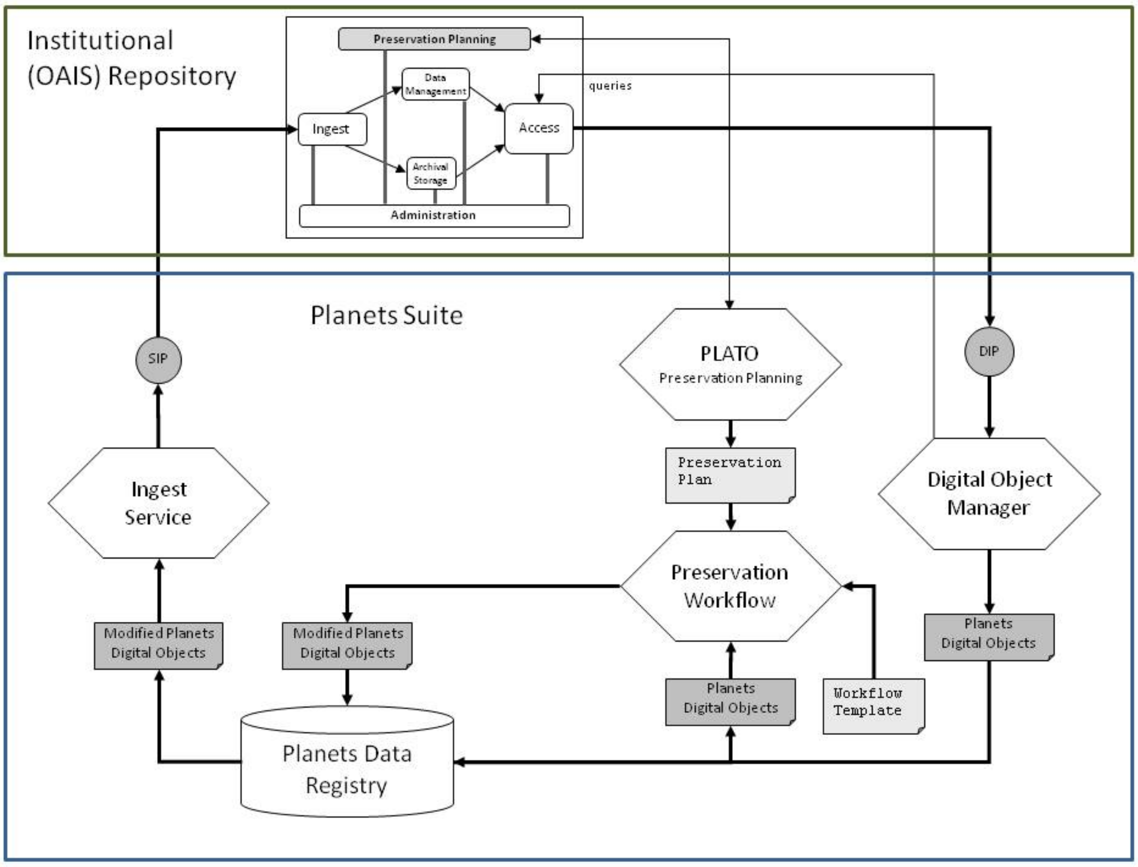

Figure 1: Planets and OAIS-compliant Repositories

It should also be mentioned that the Planets software suite is not and never was intended to be a fully OAIS[1]-compliant infrastructure. Rather, it was always assumed that Planets stakeholders would have an archive in place, whose functionality would be extended by the Planets Suite in order to carry out preservation planning and preservation actions on the digital objects within the archive (see Figure 1). This design decision has consequences that are explored in more detail in the following sections.

1.2 Variations

A critical source of technical variation between the partners was that of the choice of repository system, both in terms of variation in hardware and operating system, as well as the choice of architectural trade-offs. Most partners used bespoke storage software, based on their needs and costs, in some cases designed in-house, and in other cases in partnership with others outside of the Planets project.

This lead to perhaps the most important design decision in the project: Planets is not a repository. In contrast to other systems (e.g. RODA), we do not attempt to blend storage and preservation actions into a single system, and instead provide a well-defined framework for integrating with existing storage systems. This also reflects an expectation that an institution's repository storage system and management process will only change slowly, due to the costs associated with migrating large amounts of data safely. In contrast, the range of software tools, and the rate of exploration and investigation of different possible actions mean the preservation tools of choice are likely to change much more rapidly, even in a production setting. This flux is a natural consequence of the uncertainty around how best to preserve access to digital objects over time.

Another important source of variation between the partners was the requirements for metadata support. This includes variation in the models and metadata entities that different institutions wish to store, as well as the forms in which they wish to encode them. In general, the metadata held by the partner institutions reflects the internal evolution of metadata management and practice, and can vary within each institution between different collections, depending on the needs of the designated community as well as the institutional context. However, it was recognised that while the full extent of metadata requirements could not be captured, some core metadata could probably be meaningfully shared (e.g. Dublin Core). Therefore, the Planets software needs to be agnostic towards metadata requirements, allowing local variations to be catered for while encouraging metadata standardisation where possible and appropriate.

1.3 Commonalities

Despite the variation in metadata, the variation in the technical properties of the actual content items was perhaps surprisingly low. The range of content items held was analysed (Gap analysis: a survey of PA tool provision, PA/2 D3), and it was found that the vast majority of content was covered by around 20 file formats (mostly TIFF, JPG and PDF).

This made it possible to agree on a Digital Object concept, at least in terms of a single bitstream with associated metadata records, making exchange of digital objects possible via a standardised serialization. However, the gap analysis also indicated that the content distribution contains a very long tail of different formats and composite (multi-bitstream)

1 Open Archival Information System standard (OAIS), ISO 14721:2003

types. Therefore, the Digital Object model had to be easy to extend to new digital object types, and be able to describe composite digital objects as necessary.

As well as content types, the core actions that the partners wished to perform we also found to be shared across institutions. For example, identification, characterisation, migration and other preservation 'verbs' were already in use in each of the partners, and it was recognised that it should be possible to standardise these actions and integrate the tools that implement them. The preservation workflow that an institution chooses to employ may vary depending on the context, but the tools and services used to implement that workflow should be shared.

1.4 Consequences

To summarise, the types of digital objects vary weakly between institutions, whereas the metadata requirements vary strongly, and the storage systems and management procedures required vary massively and are represent a further, distinct field of research. Many of these differences are fundamental to the needs and requirements of each institution, requiring us to separate the institutionally-dependent needs from the needs common to all institutions.

These observations lead to the critical guiding principle of the development - *to facilitate, not dictate*. The Planets approach only enforces enough standardization to facilitate the exchange of data necessary to implement a functional digital preservation system. The Planets approach does not force a particular institution to organise its storage or metadata in a particular way, or limit the range of tools, approaches, or evaluation methodologies that an institution might employ. However, it does attempt to provide a framework on which further standardisation can be built.

2. Common API

2.1 Introduction

This section describes the API of the Planets Interoperability Framework (IF). The purpose of this section is to provide a high-level introductory overview of the API. For detailed documentation consult the Javadoc API documentation and the unit tests, both included in the Planets Services (PSERV) project.

(see http://gforge.planets-project.eu/gf/project/pserv)

The Planets Interoperability Framework (IF) has three levels of access: UI, web service API and native Java API (see Figure 2).

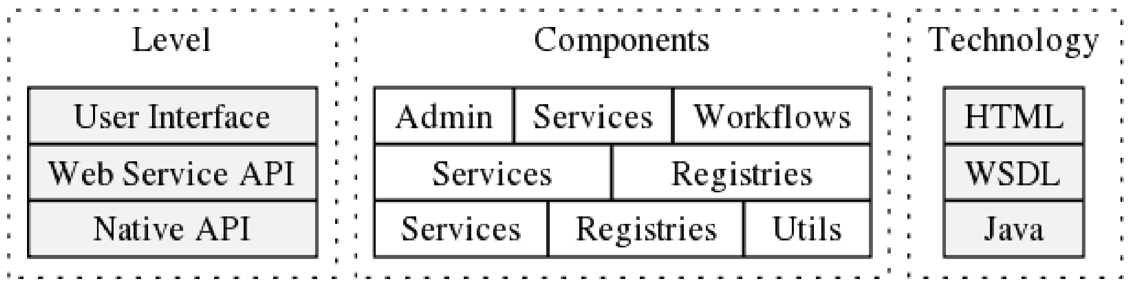

Figure 2: Planets IF overview

This section is organized around two central aspects of the Planets IF API: services and data. In each of these sections, related concepts and API elements are covered, in particular the central interfaces and data types, the service registry, the data registry and the format registry.

2.2 Services

The Planets IF takes a Java-first approach to web services. This means the WSDL is generated from Java Interfaces containing web service annotations. Therefore, the Planets IF can be used both as a platform-independent web service framework and as a plain Java library for digital preservation systems and service development.

2.2.1 Service Interfaces

The different kinds of Planets services are defined as Java Interfaces. Each preservation action verb has a corresponding Java Interface, e.g.

```
Migrate, Validate, Identify, Characterise, Compare, Modify
```

Additionally, there are some specialized versions of these Interfaces and some additional, less common interfaces (see the `eu.planets_project.services` package).

A service implementation written in Java implements the desired Interface and defines itself as a web service using the `@WebService` annotation with an `endpointInterface` attribute. This allows the service to reuse all web service specific settings from the Interfaces, without declaring them itself.

For Java clients, using JAX-WS (see https://jax-ws.dev.java.net/) enables usage of the web service without interfering with SOAP or a WSDL directly, using a proxy object that appears to the client as a normal object implementing the given Interface (Migrate, Validate, etc.).

A non-Java client can generate stubs from the WSDL that is generated from the service by JAX-WS in the way specified in the Interfaces. This way, Planets services can be accessed in a language-independent way.

Figure 3 provides an overview of the Planets IF services API.

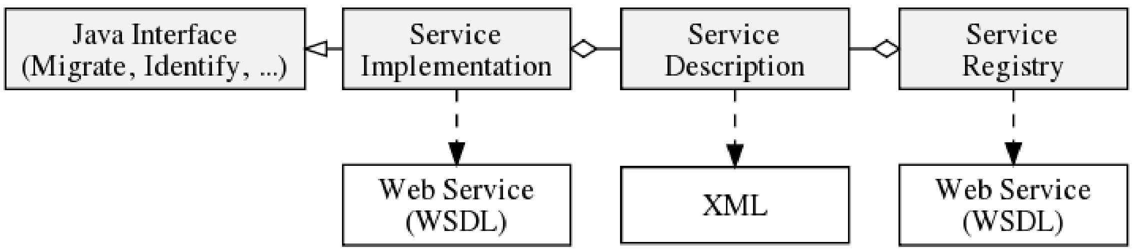

Figure 3: Planets IF services Java (top) and web service (bottom) API

2.2.2 Service Responses

All Planets Interfaces define a method corresponding to the verb they implement (e.g. migrate, validate, identify), which returns a result, whose type is determined by the verb (e.g. `MigrateResult, ValidateResult, IdentifyResult`).

Result types are typically made of the verb-specific result (e.g. a digital object for migration, a format for identification, etc.) and a general service report. A service report consists of a type (info, error, warn), a status (success, installation error, tool error) and a message:

```
ServiceReport report = new ServiceReport(Type.INFO, Status.SUCCESS, message;
```

2.2.3 Service Implementation

The following sections provide an overview both on implementing Planets services (being Planets services) and calling Planets service implementations (using Planets services).

2.2.3.1 Being Planets services

To make functionality usable as Planets services (and have them registered in a Planets service registry) the best way to go is to implement the most suitable Planets preservation verb Interface (e.g. `Migrate, Validate, Identify, Characterise, Modify,` ...). This will not only allow the services to be registered in a Planets service registry (see Figure 4), but will also allow preservation workflows querying the service registry (e.g. for a migration service) to discover the service.

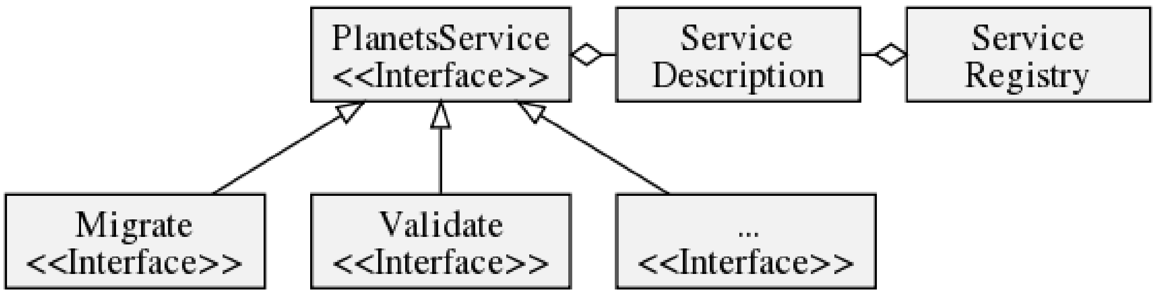

Figure 4: Planets IF services: Java classes and interfaces

If implementing one of the preservation verb Interfaces is not feasible and automatic support for discovery by workflows is not required, a service could also directly implement the `PlanetsService` Interface. This allows the service to be stored in a Planets service registry. It can be queried based on all the properties set in the service description, but will not automatically be interoperable on a workflow level with Planets services (see Figure 3).

Planets services are only registered indirectly in a service registry, using a `ServiceDescription` object (see Figure 3 and details in the following section). These objects can be serialized to XML. To register a non-Java service in a Planets service registry, an XML representation conforming to the Planets service description schema can be generated and stored in a Planets service registry.

2.2.3.2 Using Planets services

As described above, on the basic level Planets services are Java classes implementing certain Java Interfaces. If you want to access Planets functionality on the Java platform without working in a web service environment, Planets can be seen as a normal Java API, i.e. you can just instantiate classes, e.g:

```
Migrate jtidy = new JTidy();
```

See the API documentation for further details.

If you want to access remote Planets web services from Java, you can use JAX-WS to hide the SOAP layer and retrieve a proxy object from a server that will conform to the interface (i.e. you will work with an instance of a class that implements the interface, e.g. Migrate:

```
URL wsdl = new URL("http://127.0.0.1:8080/pserv-pa-jtidy/JTidy?wsdl");
Migrate jtidy = ServiceUtils.createService(Migrate.QNAME, Migrate.class, wsdl);
```

To access Planets services from non-Java platforms, you can either generate stubs from the WSDL exposed for the service or directly create SOAP messages conforming to the web service schemas (for service descriptions, digital objects, etc).

2.3 Service Descriptions

Each Interface described above extends the `PlanetsService` Interface, which defines a `describe()` method. This method returns a service description containing service metadata like input format, tool name and version, service provider, etc. The XML representations of these service descriptions are used to register and look up services in the Planets service registry (cf. Figure 4).

A flexible *query by example* mechanism (cf. next section) allows for service lookup based on various service attributes (see Figure 5 for all available service description attributes).

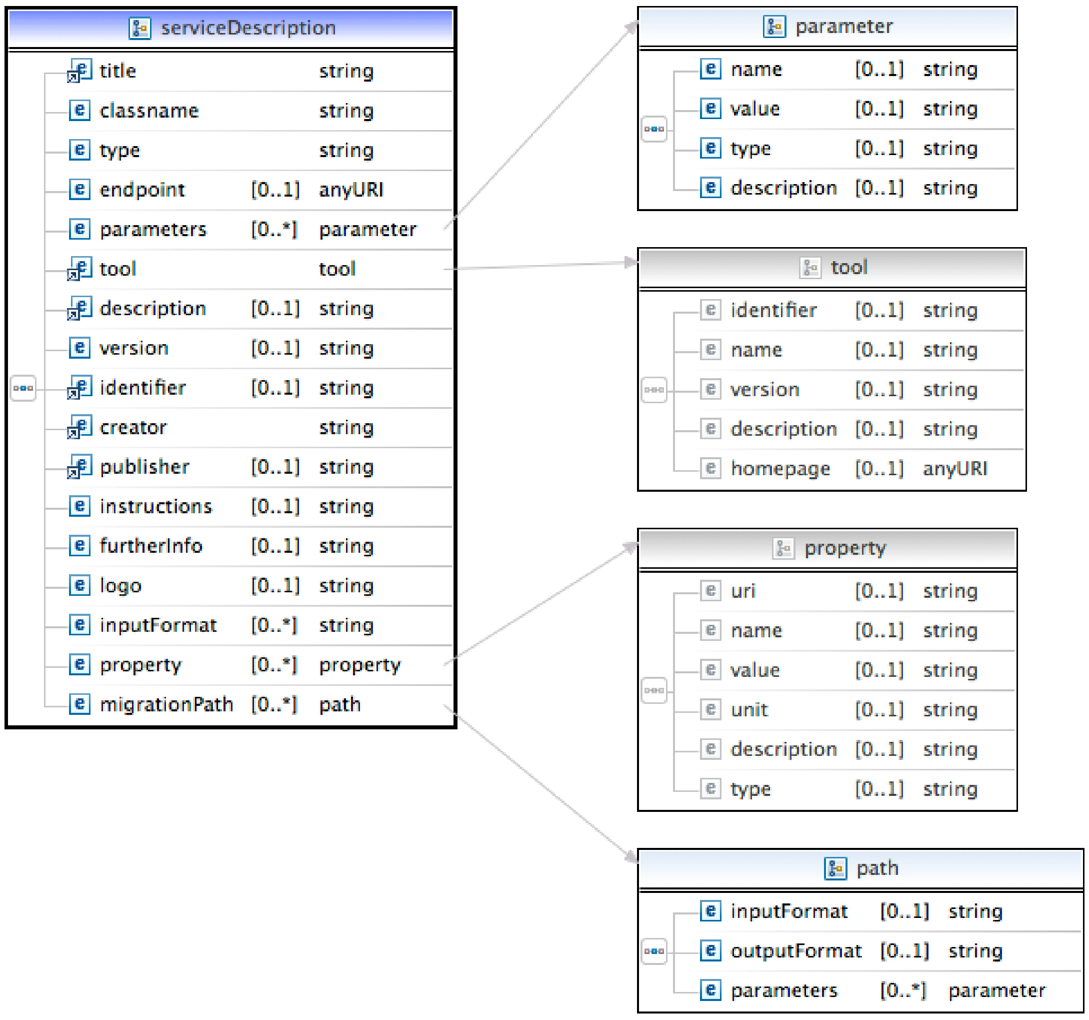

Figure 5: Service description attributes (used for queries by example in the service registry)

2.4 Service Registry

The Planets service registry enables users and service providers to look up, publish and manage information about Planets services. This information can be used to dynamically select and invoke simple services, as well as to reuse them as part of choreographed complex workflows.

The service registry code is available from the Planets Services (PSERV) project (see http://gforge.planets-project.eu/gf/project/pserv) (in IF/servreg). The service registry can be accessed from the IF administration interface and as a Java or SOAP-based web service API.

2.4.1.1 Java API

The local service registry can be accessed via the `ServiceRegistryFactory` class:

```
ServiceRegistry registry = ServiceRegistryFactory.getServiceRegistry();
```

To connect to a remote service registry, supply the location of the WSDL to the factory method.

Using the service registry instance, `ServiceDescription` objects can be registered and the registry can be queried. Queries are submitted using a *query by example* mechanism: a `ServiceDescription` object describing the services to find is created and passed to the query method, e.g. to find identification services by name:

```
registry.query(
  new ServiceDescription.Builder("DROID", Identify.class.getName()).build());
```

The given values of the service description can be matched using different modes: exact (default), wildcard, and regular expression matching, e.g. for finding identification services by name and using wildcard matching:

```
registry.queryWithMode(new ServiceDescription.Builder(
     "DROID*", Identify.class.getName()).build(), MatchingMode.WILDCARD);
```

Figure 4 shows the structure of the `ServiceDescription` objects used as examples in the queries. Any combination of these attributes can be set in the query example, allowing flexible and customizable service queries (e.g. *find migration services that can migrate PNF to TIFF based on the ImageMagick tool*).

The query methods of the service registry return a list of `ServiceDescription` objects. These service descriptions contain the information required to instantiate a service (in particular, the service endpoint).

The IF offers convenience API to instantiate a service from a service description in a Java environment, e.g. for an identification service from the example above:

```
Identify service = ServiceUtils.createService(serviceDescription);
```

See `RemoteServiceCreationTests` in the code repository for a complete example on using the service creation utility methods.

2.4.1.2 SOAP API

The mechanisms and API described above (query by example, optional matching mode) can be accessed via SOAP, passing the sample service description in its XML representation.

2.5 Data

2.5.1 Digital Objects

The central noun involved in using the Planets IF API is the *digital object*. The digital object model is described in detail in section 3.

2.5.2 Formats

Another central noun in the Planets IF API is the format, e.g. to specify the format a digital object should be migrated to (e.g. *migrate a digital object to PNG*) or the format of a file to validate (e.g. *validate that a digital object is a PNG*).

Formats are represented as URIs in the Planets IF API. The URIs can specify a PRONOM ID (see http://www.nationalarchives.gov.uk/PRONOM/), a file extension, or a MIME type.

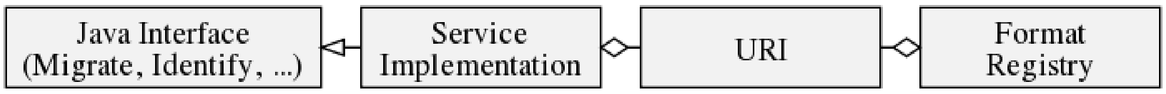

Figure 6: Planets IF formats Java API

2.5.3 Format Registry

The format registry enables access to and creation of the various format-URIs used in the Planets IF API. A registry instance can (as for the other registries) be obtained from the corresponding factory (which again enables us to hide the actual registry implementation behind the API):

```
FormatRegistry registry = FormatRegistryFactory.getFormatRegistry();
```

Given the registry, we can create format URIs for PRONOM IDs, MIME types or file extensions, e.g.:

```
URI puid = registry.createPronomUri("fmt/13");
```

The format registry also provides ways to map the different format types (PRONOM, MIME, extension) onto each other, e.g.:

```
Set<String> extensions = registry.getExtensions(puid);
```

2.6 Usage Samples

Implementations of services of all the different Interfaces (`Migrate, Validate, Identify`, etc.), including client sample usage, usage of digital objects, service descriptions and usage of the service, data and format registries, as well as complete documentation and sample usage for the classes and Interfaces described above can be found in the Planets Services (PSERV) project (see http://gforge.planets-project.eu/gf/project/pserv).

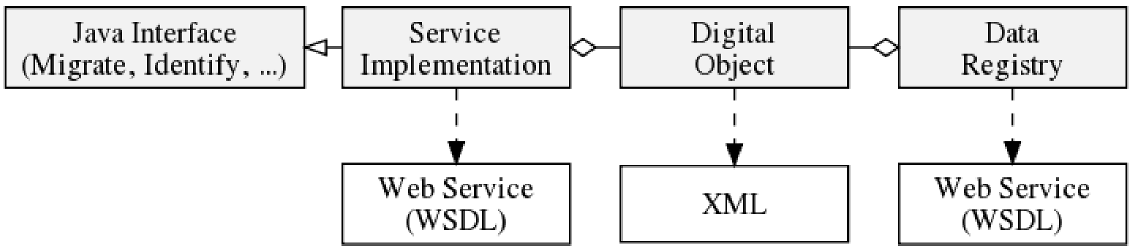

Figure 7: Planets IF digital object Java (top) and web service (bottom) API

3. Digital Object Model

3.1 Digital Objects

The central noun involved in using the Planets IF API is the *digital object*. A digital object represents a single byte stream involved in a preservation action (e.g. *identify a digital object*). Figure 6 provides an overview of the Planets IF digital object API.

3.1.1 Content Representation

To allow streaming of large files over web services, a digital object's content can be created in two conceptually different ways: *by value* (the content is actually embedded in the XML representation of the digital object and thus in the SOAP message, too) or *by reference* (the content will be streamed as an attachment to the SOAP message).

Embedded content by value can be created from a byte array, a file, or an input stream:

```
new DigitalObject.Builder(Content.byValue(bytes)).build();
```

Streamable content by reference can be created from a URL, a file, or an input stream:

```
new DigitalObject.Builder(Content.byReference(url)).build();
```

In whatever way the content has been created, it can be read in a uniform way:

```
InputStream stream = digitalObject.getContent().getInputStream();
```

When passed over web services, content by reference has the ability to be attached to the SOAP message and be streamed from the client to the server and vice versa, based on web service interoperability standards implemented by the METRO web service stack (see http://java.sun.com/webservices/). This avoids storing the entire digital object in the RAM and therefore enables the transmission of digital objects that are larger than the available RAM on the client or server side.

3.1.2 Interfaces and Builders

The central entities of the Planets API (`DigitalObject`, `ServiceDescription`, etc.) are implemented as immutable classes, which are created using builders.

The usage of a builder can be seen in the example illustrating the content representation above: a minimal digital object consists of nothing but its content. If we want to set additional attributes of the digital object (e.g. a unique ID), these are set on the builder before the object is built:

```
  new DigitalObject.Builder(Content.byValue(bytes)).permanentUri(id).build();
```

By setting these attributes on the builder, the resulting digital object instance can be immutable and thus be used and shared freely, also in concurrent computing setups. At the same time, combining an Interface with a builder allows for the actual implementation class to be hidden behind the API (e.g. to be changed or swapped out after releasing the API). Having a `DigitalObject` Interface also allows third party implementations of the Planets digital object model.

3.2 XML Serialization

To allow passing of digital objects via web services, a JAXB XML adapter defines a standard `DigitalObject` implementation (`ImmutableDigitalObject`) to be used. This implementation, however, is hidden from a Java API consumer and only exposed as a WSDL.

Besides supporting XML serialization for web service usage, digital objects can also be stored as XML directly via the API:

```
  String xml = digitalObject.toXml();
```

Given such an XML representation, the digital object instance can be instantiated directly via the API:

```
  DigitalObject object = new DigitalObject.Builder(xml).build();
```

The XML serialization schema of the digital object is described in Figure 8 (digital object) and 9 (structure of the contained event properties).

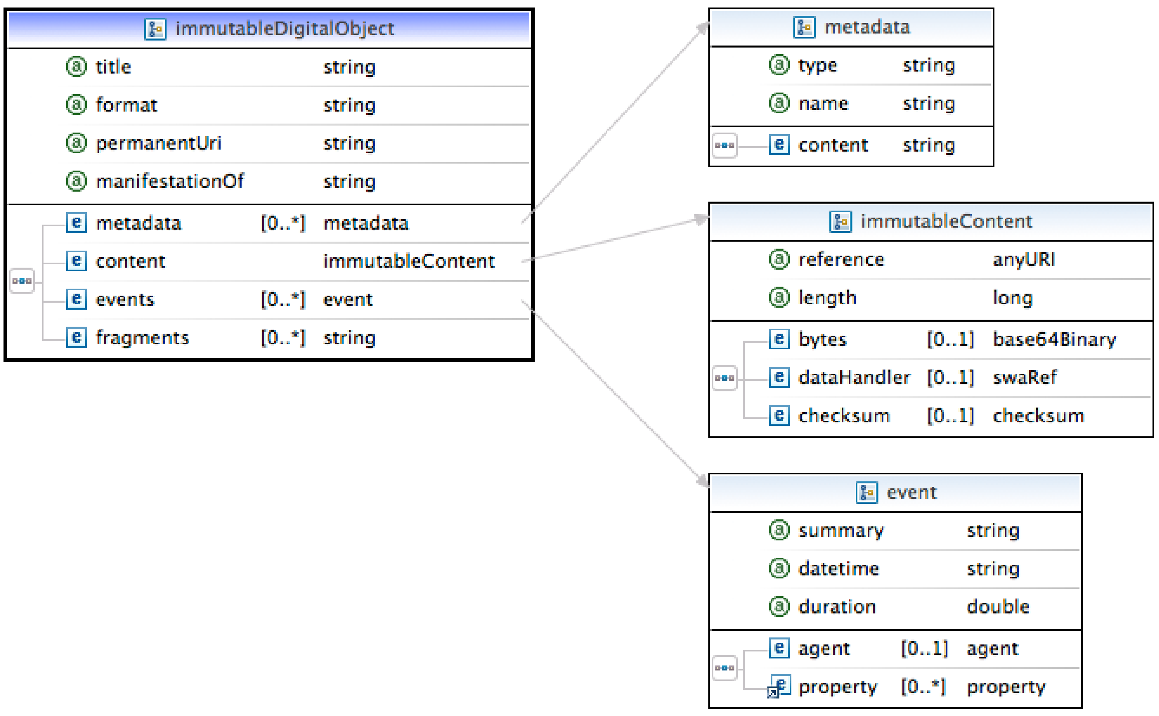

Figure 8: Digital Object XML Schema Definition

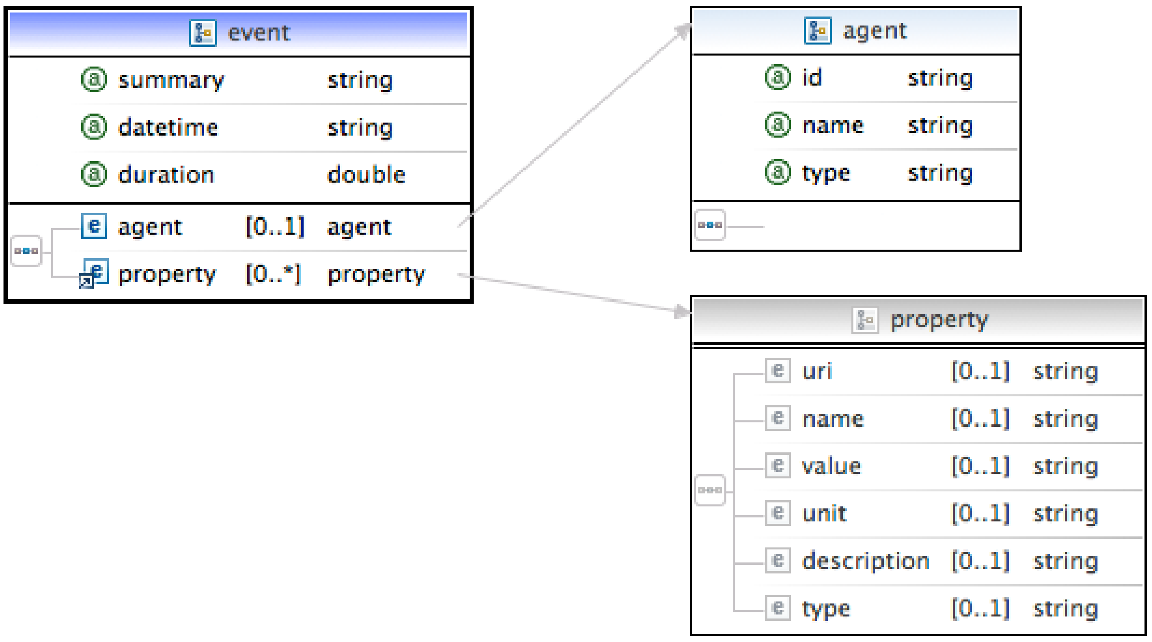

Figure 9: Digital Object Event XML Schema Definition

3.3 Data Registry

The data registry enables storage and retrieval of digital objects. The `DigitalObjectManager` Interface can be implemented for different concrete storage solutions.

3.3.1 Java API

Like the `ServiceRegistry`, a `DataRegistry` is instantiated using its factory:

```
  DataRegistry registry = DataRegistryFactory.getDataRegistry();
```

The data registry can be used to store digital objects under an ID, represented as a URI:

```
  registry.store(uri, digitalObject);
```

Digital objects can be retrieved from the data registry using the given ID:

```
  DigitalObject digitalObject = registry.retrieve(uri);
```

3.3.2 SOAP API

Like the service registry, the Planets data registry is also available as a SOAP-based web service, which uses the XML serialization mechanism of digital objects described above and allows language-independent access to a Planets data registry.

4. Workflow Engine

4.1 Introduction

The Planets Workflow system is integrated with most other building blocks of the Planets Interoperability Framework (IF) and is one of its core components. It provides both interfaces and utilities which allow domain experts to incorporate Planets Services in complex workflow operations, taking into account domain specific business logic (as decision making), as well as components to configure, interpret, and execute these models in a controlled environment.

4.2 Components and Interaction

Generally speaking the Planets Workflow Execution Engine (WEE) component exposes its functionality through two web-services / stateless-session-beans which provide the application's public interface:

- *WorkflowTemplateRegistry* allows browsing, registering and retrieving Workflow Templates

- *WorkflowExecutionManager* allows submitting workflows with payload and polling for the execution's status, progress and results.

Additionally, there is a message-driven-bean: the *WorkflowExecutionEngine* - the actual batch job processor.

4.2.1 Workflow Template and Utilities

Workflows which are processed on the Planets Framework need to implement the `eu.planets_project.ifr.core.wee.api.workflow.WorkflowTemplate` interface. It defines

- Planets Service type interfaces which are used (e.g. `Identify, Migrate, Compare`) but not the actual service instances (e.g. Droid or ImageMagick)
- The structure and decision making process every single Digital Object within a workflow has to pass through (e.g. branching, looping, exception handling, data and service flow, event handling, etc.)

It is also responsible for depositing results within Planets data repositories, documenting actions in a structured and traceable manner (by using the `WorkflowResult` and `WorkflowResultItem` logging objects) and most often will be reflecting events on the Digital Object's data model itself.

Creating a workflow template is straight-forward and simple. A compliant Java class implements the *WorkflowTemplate* interface and extends the provided *WorkflowTemplateHelper* class, which provides an implementation of all the underlying commonly used convenience functionality like data access, etc. as well as all the required "magic" to dynamically build up, configure and instantiate a `WorkflowInstance` through reflection at runtime. A `WorkflowTemplateProvider` therefore only needs to implement the methods `execute`() and `describe()`.

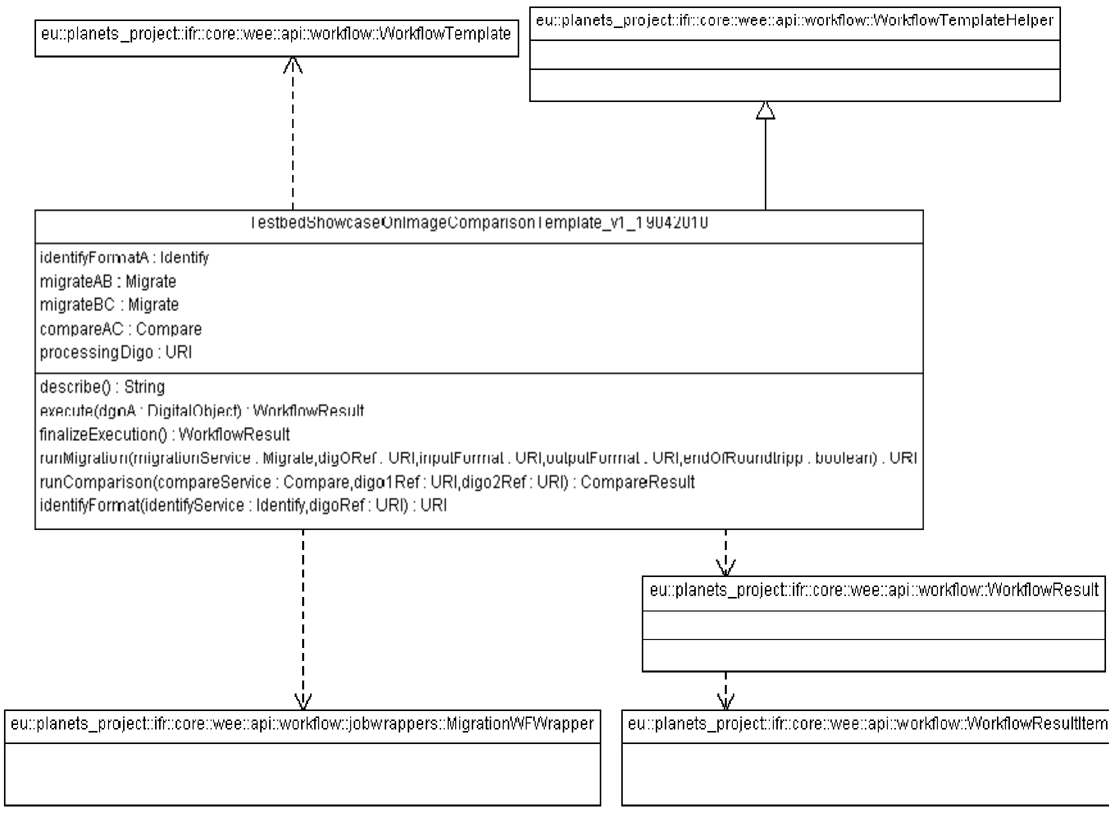
eu::planets_project::ifr::core::wee::api::workflow::WorkflowTemplate
eu::planets_project::ifr::core::wee::api::workflow::WorkflowTemplateHelper
TestbedShowcaseOnImageComparisonTemplate_v1_19042010
identifyFormatA : Identify
migrateAB : Migrate
migrateBC : Migrate
compareAC : Compare
processingDigo : URI
describe() : String
execute(digoA : DigitalObject) : WorkflowResult
finalizeExecution() : WorkflowResult
runMigration(migrationService : Migrate,digORef : URI,inputFormat : URI,outputFormat : URI,endOfRoundtripp : boolean) : URI
runComparison(compareService : Compare,digo1Ref : URI,digo2Ref : URI) : CompareResult
identifyFormat(identifyService : Identify,digoRef : URI) : URI
eu::planets_project::ifr::core::wee::api::workflow::WorkflowResult
eu::planets_project::ifr::core::wee::api::workflow::jobwrappers::MigrationWFWrapper
eu::planets_project::ifr::core::wee::api::workflow::WorkflowResultItem

Figure 10: Sample workflow template implementation class diagram

All Services that implement a Planets Service interface are allowed within a workflow template. There are only three restrictions (one implicit and two explicit ones) that a template provider needs to be aware of to provide a valid implementation.

The first one is a naming convention: The IDs used within the `WorkflowConfiguration` (xml) must corresponds to the object's variable (field) name within the template class e.g. `<service id="`**`identify1`**`">` and `private eu.planets_project.services.identify.Identify` **`identify1`**`;`

The other restrictions are enforced by the `WorkflowFactory`, which validates that the `ServiceTemplate` interface is implemented, and checks that all declared Service interfaces within the workflow are within the range of supported Planets service types.

4.2.2 Workflow Configuration

An XML representation is used to configure and instantiate a particular workflow template so that it's executable by the WEE subsequently. These XML documents are intended to serve as 'Executable Preservation Plans' - which are generated and exchanged by the Planets applications as Testbed or Plato – and contain recommendations on service instances and their corresponding parameter configurations while the payload (i.e.

DigitalObject references) is handed over separately to fulfil the requirements on the underlying process flow.

The `WorkflowFactory` is responsible for unmarshalling the provided workflow XML configuration by using JAXB generated objects of the planets-wdt schema (which can be found at http://www.planets-project.eu/private/planets-ftp/WP_IF/IF5/wee/v1Feb2009/planets_wdt.xsd).

The first snippet in the appendix presents a sample configuration for instantiating the given `eu.planets_project.ifr.core.wee.impl.templates.TestbedShowcaseOnImageComparisonTemplate` workflow template with specific service endpoints and their corresponding parameter configurations.

4.2.3 Workflow Factory and Workflow Instance

The `WorkflowFactory` class contains a single public method which takes the JAXB Java representation of an XML workflow configuration together with the payload to process the given workflow upon and returns a `WorkflowInstance` object which is processable by the batch execution backend.

```
public static WorkflowInstance create(WorkflowConf wfConf,
List<DigitalObject> digos) throws Exception
```

The process steps involved in returning an immutable `WorkflowInstance` object include:

- querying and fetching the requested java workflow source file from the registry
- validating, compiling, jar-ing and classloading workflow templates on the fly
- dynamically reflecting on the workflow template's fields for
- building and initializing proxies for the passed service endpoints and storing their service parameter configuration in a `WorkflowContext` object.
- including the payload (i.e. data registry references in terms of DigitalObjects) to invoke the workflow upon.

The returned *WorkflowInstance* object can be submitted to the queue and executed on the Planets WorkflowExecutionEngine.

4.2.4 WorkflowTemplateRegistry

As mentioned in the section above, the workflow execution system allows to browse, retrieve and submit WorkflowTemplates. The following operations are supported:

- *getAllSupportedQNames*: returns a list of all fully qualified workflow template names (`QName`) which have been registered on the system and therefore are ready to use.

- *getWFTemplate*: returns the source (`WorkflowTemplate.java` source file) of a requested `QName`. As the business logic of the underlying workflow is modelled in

Java, it might be necessary to retrieve and explore the source to fully understand a template's behaviour.

- *registerWorkflowTemplate*: Expects a fully qualified name of the submitted class. e.g. 'eu.planet_project.ifr.core.TemplateName.java' as well the byte representation of the Java source file.

4.2.5 WorkflowExecutionManager

A web service / stateless session bean is used by the WEE to request submitted workflow instances, notify back upon results and progress made, and as an interface to the end users or user applications for requesting the status, position, progress or results of a submitted workflow execution.

- *submit workflow*: the job submission service accepts a of list of Planets Digital Objects (or Digital Object references) which contain the payload the workflow is invoked upon, the fully qualified name of the (already previously registered) workflow template, together with the specific XML workflow config holding the workflow template's configuration to apply. This method returns a job-ticket (UUID) which can subsequently be used for on the status and its results

- querying on a job's *status*, *position* in queue or *progress*

- *retrieving execution results* (or intermediate results in case the execution is still processing)

4.2.6 Batch Processor

The WEE job processor is exposed as a Java message-driven bean (MDB) with an javax.ejb.ActivationConfigProperty *maxSession* configuration of one. This guarantees that only a single instance of the MDB is running and therefore only one job at a time is being executed.

Workflow jobs are submitted to the javax.jms.Queue by UUID. Their corresponding `WorkflowInstance` objects containing the processing logic and data payload are requested by the batch processor from the `WorkflowManager` when operated upon. Submitted `WorkflowInstances` are processed in a FIFO order and executed by the blocking *onMessage* operation of the implementing class.

The workflow is processed per DigitalObjects and intermediate results as well as progress information is reported back to the WorkflowManager. The second code snippet in the appendix shows the batch processor’s *onMessage* implementation.

4.2.7 Logging Workflow Results

In order to trace and exchange information which is being created during a workflow execution in a structured way the classes `eu.planets_project.ifr.core.wee.api.workflow.WorkflowResult` and `WorkflowResultItem` are provided.

They allow to trace changes per digital object throughout the entire workflow i.e. all service operations which took place on a given object, extracted outcomes (as identification information or migrated output object references), general process information on the workflow itself (e.g. did the workflow succeed properly on all steps of the workflow, workflow execution time) and service calls (e.g. service reports, -endpoint and –description), together with any general log information. These classes additionally write to a custom `ReportingLog` which allows debugging in case of a system crash or failure.

4.2.8 Utility Wrappers

Although workflows are specific in their individual functionality there is a set of basic operation patterns that are common to almost all cases. For those operations the WEE contains so called ‘job wrappers’, which can easily be reused.

- Migration Utility Wrapper: A provided migration workflow utility is able to take care of the most common migration behaviour as logging proper workflow result statements, persisting digital objects in the default data registry and returning objects as shared data registry URIs as well as creating default events of the preservation action for the digital object at hand.
- Log-Reference Creator Utility Wrapper: Assembles all created log-statements and creates a globally accessible workflow log file for the processed template.

4.3 Usage Samples

Implementations of existing workflow templates have been provided throughout the project and are used by applications as the Planets Testbed, Plato and the Workflow Design Tool.

The sample workflow implementation in the appendix is provided by the Planets Testbed for performing a migration from format A to B and B to C, where format A (and output of C) is automatically determined by a preceding identification service. For post-migration analysis this workflow calls a comparison service to check on the similarity between objects A and C and documents that information within the workflow’s results.

Further workflow examples and their corresponding sample configuration are available within the Planets IF_SP SVN repository (see http://gforge.planets-project.eu/gf/project/if_sp/components/wee/src/main/resources/eu/planets_project/ifr/core/wee/impl/templates).

5. Appendix

5.1 Workflow Configuration

```xml
<?xml version="1.0" encoding="UTF-8"?>
<workflowConf xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="planets_wdt.xsd">
  <template>
    <class>eu.planets_project.ifr.core.wee.impl.templates.TestbedShowcaseOnImageComparisonTemplate_v1_05022010</class>
  </template>
  <services>
  <service id="identifyFormatA">
      <endpoint>http://testbed.planets-project.eu:80/pserv-pa-sanselan/SanselanIdentify?wsdl</endpoint>
    </service>
    <service id="migrateAB">
      <endpoint>http://testbed.planets-project.eu:80/pserv-pa-imagemagick/ImageMagickMigrate?wsdl</endpoint>
      <parameters>
        <param>
          <name>planets:service/migration/input/migrate_to_fmt</name>
          <value>planets:fmt/ext/jpeg</value>
        </param>
        <!-- Specify 5 for JPEG Compression, comprQuality 80 percent -->
        <param>
          <name>compressionType</name>
          <value>5</value>
        </param>
        <param>
          <name>compressionQuality</name>
          <value>80</value>
        </param>
      </parameters>
    </service>
    <service id="migrateBC">
      <endpoint>http://testbed.planets-project.eu:80/pserv-pa-imagemagick/ImageMagickMigrate?wsdl</endpoint>
      <parameters>
        <param>
          <name>planets:service/migration/input/migrate_from_fmt</name>
          <value>planets:fmt/ext/jpeg</value>
        </param>
        <!-- Specify 5 for JPEG Compression, comprQuality 80 percent -->
        <param>
          <name>compressionType</name>
          <value>5</value>
        </param>
        <param>
          <name>compressionQuality</name>
          <value>80</value>
        </param>
      </parameters>
    </service>
    <service id="compareAC">
      <endpoint>http://testbed.planets-project.eu:80/pserv-pa-java-se/JavaImageIOCompare?wsdl</endpoint>
    </service>
  </services>
</workflowConf>
```

5.2 Batch Processor

```java
/* @see javax.jms.MessageListener#onMessage(javax.jms.Message) */
public void onMessage(Message m) {
  // check if message got redelivered before doing any processing
  try {
    if (m.getJMSRedelivered()) {
      log.debug("WorkflowExecutionEngine: onMessage: re-receive message from the queue. Not processing 
it");
      return;
    }
  } catch (JMSException e) {
    log.debug(e);
  }
  // 1) get the WEEManager instance - required in the same JVM
  WeeManager weeManager = WeeManagerImpl.getWeeManagerInstance();
  // 2) extract the Message's payload
  WorkflowInstance wf = null;
  UUID uuid = null;
  try {
    TextMessage msg = null;
    if (m instanceof TextMessage) {
      msg = (TextMessage) m;
      log.debug("WorkflowExecutionEngine: received ObjectMessage at timestamp: "
          + msg.getJMSTimestamp());
    }
    // for ObjectMessages: uuid = UUID.fromString(msg.getStringProperty("UUID"));
    uuid = UUID.fromString(msg.getText());
    /* WorkflowInstance object cannot be Serialized due to the
     * org.jboss.ws.core.jaxws.client.ClientProxy it contains that cannot be serialized. Therefore
     * a callback to the weeManager for fetching this object is performed. Not possible: wf =
     * (WorkflowInstance)msg.getObject(); */
    wf = ((WeeManagerImpl) weeManager).getWorkflowInstance(uuid);
  } catch (Exception e) {
    log.error("WorkflowExecutionEngine: error receiving message workflow payload or UUID", e);
    if (uuid != null) {
      weeManager.notify(uuid, WorkflowExecutionStatus.FAILED);
    }
    return;
  }
  // set status for the workflow inISRUNNING
  weeManager.notify(uuid, WorkflowExecutionStatus.RUNNING);
  // 3) executeWorkflow and get WF Result
  WorkflowResult ret = wf.initializeExecution();
  try {
    log.debug("WorkflowExecutionEngine: start executing wf ID: " + wf.getWorkflowID());
    // EXECUTES THE WF INSTANCE
    List<DigitalObject> payload = wf.getData();
    int count = 1;
    for (DigitalObject digo : payload) {
      // process the payload item by item - workflowResult appends individual log items
      ret = wf.execute(digo);
      count += 1;
      int progress = (100 / payload.size()) * count;
      weeManager.notify(uuid, ret, WorkflowExecutionStatus.RUNNING, progress);
    }
    ret = wf.finalizeExecution();
    log.debug("WorkflowExecutionEngine: completed executing wf ID: " + wf.getWorkflowID());
  } catch (Exception e) {
    log.error("WorkflowExecutionEngine: error running Workflow.execute()", e);
    // set WeeManagerstatus 'failed'
    weeManager.notify(uuid, WorkflowExecutionStatus.FAILED);
    return;
  }
  // 4) call WEEManager.notify to report back results and the status
  weeManager.notify(uuid, ret, WorkflowExecutionStatus.COMPLETED);
}
```

5.3 Workflow Usage Sample

```
public class TestbedShowcaseOnImageComparisonTemplate_v1_05022010 extends WorkflowTemplateHelper
    implements WorkflowTemplate {
  private Identify identifyFormatA;
  private Migrate migrateAB;
  private Migrate migrateBC;
  private Compare compareAC;
  private URI processingDigo;
  /* (non-Javadoc)
   * @see eu.planets_project.ifr.core.wee.api.workflow.WorkflowTemplate#describe() */
  public String describe() {
    return "This template performs a A-B and B-C migration action, where format of A (and output of C)
is automatically "
        + "determined by an Identification service."
        + "For post-migration-analysis this workflow calls a comparison service to check on similarity
of A and C (e.g. in terms of PSNR) and documents that information. ";
  }

  @Override
  public WorkflowResult initializeExecution() {
    this.getWFResult().setStartTime(System.currentTimeMillis());
    return this.getWFResult();
  }

  /* (non-Javadoc)
   * @see eu.planets_project.ifr.core.wee.api.workflow.WorkflowTemplate#execute() */
  @SuppressWarnings( "finally" )
  public WorkflowResult execute(DigitalObject dgoA) {
    // document all general actions for this digital object
    WorkflowResultItem wfResultItem = new WorkflowResultItem(dgoA.getPermanentUri(),
        WorkflowResultItem.GENERAL_WORKFLOW_ACTION, System.currentTimeMillis(),
        this.getWorkflowReportingLogger());
    this.addWFResultItem(wfResultItem);
    wfResultItem.addLogInfo("working on workflow template: " + this.getClass().getName());
    // start executing on digital ObjectA
    this.processingDigo = dgoA.getPermanentUri();
    try {
      // run a pre-Identification service on A to determine it's format
      wfResultItem.addLogInfo("starting identification A");
      URI formatA = identifyFormat(identifyFormatA, dgoA.getPermanentUri());
      wfResultItem.addLogInfo("completed identification A");
      // Migrate Object round-trip
      wfResultItem.addLogInfo("starting migration A-B");
      URI dgoB = runMigration(migrateAB, dgoA.getPermanentUri(), formatA, null, false);
      wfResultItem.addLogInfo("completed migration A-B");
      wfResultItem.addLogInfo("starting migration B-C");
      URI dgoC = runMigration(migrateBC, dgoB, null, formatA, true);
      wfResultItem.addLogInfo("completed migration B-C");
      // compare the object's A and C
      wfResultItem.addLogInfo("starting comparison A-C");
      runComparison(compareAC, dgoA.getPermanentUri(), dgoC);
      wfResultItem.addLogInfo("completed comparison A-C");
      wfResultItem
          .addLogInfo("successfully completed workflow for digitalObject with permanent uri:"
              + processingDigo);
      wfResultItem.setEndTime(System.currentTimeMillis());
    } catch (Exception e) {
      String err = "workflow execution error for digitalObject #" + " with permanent uri: "
          + processingDigo;
      wfResultItem.addLogInfo(err + " " + e);
      wfResultItem.setEndTime(System.currentTimeMillis());
    }
    return this.getWFResult();
  }
  /** {@inheritDoc} */
  public WorkflowResult finalizeExecution() {
    this.getWFResult().setEndTime(System.currentTimeMillis());
    LogReferenceCreatorWrapper.createLogReferences(this);
    return this.getWFResult();
  }

  /**
   * Runs the migration service on a given digital object. It uses the MigrationWFWrapper to call
   * the service, create workflowResult logs, events and to persist the object within the JCR
   * repository
   */
  private URI runMigration(Migrate migrationService, URI digORef, URI inputFormat,
      URI outputFormat, boolean endOfRoundtripp) throws Exception {
    MigrationWFWrapper migrWrapper = new MigrationWFWrapper(this, this.processingDigo,
        migrationService, digORef, endOfRoundtripp);
    // possibly using identification service to determine the input/output format
    if (inputFormat != null) {
      migrWrapper.setInputFormat(inputFormat);
    }
    if (outputFormat != null) {
      migrWrapper.setOutputFormat(outputFormat);
    }
    // specifying the location where to store migration results
    migrWrapper.setDataRepository(DataRegistryFactory
        .createDataRegistryIdFromName("/experiment-files/executions/"));
    return migrWrapper.runMigration();
  }

  /**
   * Runs the comparison service on two digital objects
   */
  private CompareResult runComparison(Compare compareService, URI digo1Ref, URI digo2Ref)
      throws Exception {
    WorkflowResultItem wfResultItem = new WorkflowResultItem(this.processingDigo,
        WorkflowResultItem.SERVICE_ACTION_COMPARE, System.currentTimeMillis());
    this.getWFResult().addWorkflowResultItem(wfResultItem);
    try {
      // get all parameters that were added in the configuration file
      List<Parameter> parameterList;
      if (this.getServiceCallConfigs(compareService) != null) {
        parameterList = this.getServiceCallConfigs(compareService).getAllPropertiesAsParameters();
      } else {
        parameterList = new ArrayList<Parameter>();
      }
      wfResultItem.setServiceParameters(parameterList);
      wfResultItem.setStartTime(System.currentTimeMillis());
      // document the endpoint if available - retrieve from WorkflowContext
      String endpoint = this.getWorkflowContext().getContextObject(compareService,
          WorkflowContext.Property_ServiceEndpoint, java.lang.String.class);
      if (endpoint != null) {
        wfResultItem.setServiceEndpoint(new URL(endpoint));
      }
      ServiceDescription serDescr = compareService.describe();
      wfResultItem.setServiceDescription(serDescr);
      // retrieve the digital objects from their data registry location
      DigitalObject digo1 = this.retrieveDigitalObjectDataRegistryRef(digo1Ref);
      DigitalObject digo2 = this.retrieveDigitalObjectDataRegistryRef(digo2Ref);
      // now call the comparison
      CompareResult compareResult = compareService.compare(digo1, digo2, parameterList);
      wfResultItem.setEndTime(System.currentTimeMillis());
      ServiceReport report = compareResult.getReport();
      // report service status and type
      wfResultItem.setServiceReport(report);
      if (report.getType() == Type.ERROR) {
        String s = "Service execution failed: " + report.getMessage();
        wfResultItem.addLogInfo(s);
        throw new Exception(s);
      }
      // document the comparison's output
      if ((compareResult.getProperties() != null) && (compareResult.getProperties().size() > 0)) {
        wfResultItem.addLogInfo("Comparing properties of object A: " + digo1.getPermanentUri()
            + " with object B: " + digo2.getPermanentUri());
        for (Property p : compareResult.getProperties()) {
          String extractedInfo = "[name: " + p.getName() + " value: " + p.getValue() + " untit: "
              + p.getUnit() + " description:" + p.getDescription() + "] \n";
          wfResultItem.addExtractedInformation(extractedInfo);
        }
      } else {
        wfResultItem.addLogInfo("No comparison properties received");
      }
      wfResultItem.addLogInfo("comparison completed");
      return compareResult;
    } catch (Exception e) {
      wfResultItem.addLogInfo("comparison failed " + e);
      throw e;
    }
  }

  /**
   * Runs the identification service on a given digital object reference and returns the first
   * format that is found.
   */
  private URI identifyFormat(Identify identifyService, URI digoRef) throws Exception {
    WorkflowResultItem wfResultItem = new WorkflowResultItem(this.processingDigo,
        WorkflowResultItem.SERVICE_ACTION_IDENTIFICATION, System.currentTimeMillis());
    wfResultItem.setInputDigitalObjectRef(digoRef);
    this.getWFResult().addWorkflowResultItem(wfResultItem);
    // get all parameters that were added in the configuration file
    List<Parameter> parameterList;
    if (this.getServiceCallConfigs(identifyService) != null) {
      parameterList = this.getServiceCallConfigs(identifyService).getAllPropertiesAsParameters();
    } else {
      parameterList = new ArrayList<Parameter>();
    }
    // document
    wfResultItem.setServiceParameters(parameterList);
    // document the endpoint if available - retrieve from WorkflowContext
    String endpoint = this.getWorkflowContext().getContextObject(identifyService,
        WorkflowContext.Property_ServiceEndpoint, java.lang.String.class);
    if (endpoint != null) {
      wfResultItem.setServiceEndpoint(new URL(endpoint));
    }
    wfResultItem.setStartTime(System.currentTimeMillis());
    // resolve the digital Object reference
    DigitalObject digo = this.retrieveDigitalObjectDataRegistryRef(digoRef);
    // call the identification service
    IdentifyResult identifyResults = identifyService.identify(digo, parameterList);
    // document
    wfResultItem.setEndTime(System.currentTimeMillis());
    ServiceReport report = identifyResults.getReport();
    // report service status and type
    wfResultItem.setServiceReport(report);
    if (report.getType() == Type.ERROR) {
      String s = "Service execution failed: " + report.getMessage();
      wfResultItem.addLogInfo(s);
      throw new Exception(s);
    }
    // document the comparison's output
    URI ret = null;
    if ((identifyResults.getTypes() != null) && (identifyResults.getTypes().size() > 0)) {
      wfResultItem.addLogInfo("identifying properties of object: " + digo.getPermanentUri());
      for (URI uri : identifyResults.getTypes()) {
        if (ret == null) {
          ret = uri;
        }
        String extractedInfo = "[uri: " + uri + "] \n";
        wfResultItem.addExtractedInformation(extractedInfo);
      }
    } else {
      String s = "Identification failed: format not identified";
      wfResultItem.addLogInfo(s);
      throw new Exception(s);
    }
    wfResultItem.addLogInfo("Identification completed, using format: " + ret);
    return ret;
  }
}
```