EbRIM implementation with GeoNetwork and OMAR

Ergo

Architectural Design Document

Authors: Massimiliano Fanciulli

Reviewed by: Simone Gianfranceschi

Approved by: Patrizia Nencioni

Document Id:	Issue:	Revision:
ERG-ADD-3100-INT	1 19/12/2008	0 19/12/2008

Document change record

Issue	*Issue date*	*Pages effected*	*Reason for change*
1.0	19/12/2008	All	Draft version.

Distribution List

Company	Name	Function	N° of copies

Table of Content

1. Introduction

The MASS system was developed in the GSTP project with the same name in the period 2001-2003. It provides an enabling and open environment for the integration of Earth Observation and GIS services. Due to the interest the GSTP project has generated, the system was extended as part of the MASS-ENV and ISAC projects and services were integrated as part of the MASS-SER projects. The system is now operational at ESRIN and known as the "Service Support Environment" (SSE).

1.1. Purpose

This document is the **Architectural Design Document** (ADD) for SSE TOOLBOX software which is part of the Ergo project. This project is carried out for ESRIN by a consortium consisting of Intecs (Italy), Foresee Technologies (Belgium), GeoNetwork (Holland) under the lead of Intecs.
The TOOLBOX is a configurable application that helps the Service Provider to easily convert its service in SOAP based service without code development. The new Web Services can be integrated in the SSE framework.

The TOOLBOX software is composed by two applications:

- Run-time application (further referred as run-time environment or TOOLBOX RE)
- Development application (further referred as development environment or TOOLBOX DE)

This document contains an overview of the TOOLBOX run-time (chapter 4) and development (chapter 5) environments together with the design definition.

1.2. Scope

1.3. Glossary

1.3.1. Abbreviations

Acronym	Extended Form
API	Application Programming Interface
Back-end service	It is the service offered by the SP and deployed on the TOOLBOX. The back-end service communicates with the SSE via the TOOLBOX.
FTP	File Transfer Protocol
EOLI-XML	EARTHNET ONLINE XML FRONT-END.
Generic operation	An operation, which is not included in the SSE ICD.
Generic service	It is intended as service supporting operations besides those included in the SSE ICD (i.e. Search, Present, RFQ, Order). Likewise an operation, which is not included in the SSE ICD, is referred in the document as "generic operation".

Acronym	Extended Form
HTTP	Hypertext Transfer Protocol **Errore. L'origine riferimento non è stata trovata.**
ICD	Interface Control Document
JDBC	Java Database Connectivity
JSP	Java Server Pages
JVM	Java Virtual Machine
MASS	Multi-Application Support Service System
MASS-ENV	Multi-Application Support Service System Environment
NSI	New Service Integration
RB	Requirement Baseline Document
RFQ	Request for Quotation
SOAP	Simple Object Access Protocol **Errore. L'origine riferimento non è stata trovata.**. SOAP is a simple XML based protocol to let applications exchange information over HTTP and is used in the communication between the TOOLBOX run-time environment and the SSE Portal as well as in the communication between the TOOLBOX development environment and the TOOLBOX run-time environment.
SOAP Fault	Within the SOAP protocol, the SOAP Fault is an element of the SOAP messages used to carry error and/or status information within a SOAP message.
SP	Service Provider
SSE	Service Support Environment
SSL	Secure Socket Layer
SUM	Software User Manual
TOOLBOX RE	Toolbox Runtime Environment
TOOLBOX DE	Toolbox Development Environment
URD	User Requirements Document
URL	Uniform Resource Locator
Web Service	The term Web services describes a standardized way of integrating Web-based applications using the XML, SOAP, WSDL and UDDI open standards over an Internet protocol backbone. XML is used to tag the data, SOAP is used to transfer the data, WSDL is used for describing the services available and UDDI is used for listing what services are available.
WSDL	Web Services Description Language ([IR5])
XML	eXtensible Mark-up Language ([IR6])
XSLT	eXtensible Style Language Transformation ([IR8])

1.3.2. Definition of Terms

The following terms are used:

- Service: a service deployed on the TOOLBOX Runtime Environment and published on the SSE portal
- Back-end service: a service that uses the TOOLBOX Runtime Environment to communicate with the SSE.

1.4. References

1.4.1. Normative References

In case of conflict between two or more applicable documents, the higher document will prevail.

[NR1] HMA Requirement Baseline, HMA-RS-ASU-SY-0001 Issue 1, Revision 4

[NR2] Catalogue Service Inter-Accessibility Specifications, HMA-RS-SIE-EN-001, Issue 1, Revision 2

[NR3] ERGO Abbreviations, Terms and Acronyms, ERG-ACR-1100-INT Issue 1, Revision 0, 16/05/2008

[NR4] OGC Catalogue Services Specification 2.0.0 (with Corrigendum) EO Products Extension Package for ebRim (ISO/TS 15000-3) Profile of CSW 2.0, OGC 06-131, Issue 0, Revision 0.3, 18/08/2006

[NR5] OGC® Cataloguing of ISO Metadata (CIM) Using the ebRIM profile of CS-W, OGC 07-038, Issue 0, Revision 0.7, 10/05/2007

[NR6] ERGO Requirement Baseline, ERG-RB-2100-INT, Issue 1, Revision 1, 06/06/2008

Commento [MF1]: Copiati dall'ADD di SAS, da ricontrollare

1.4.2. Informative references

The following documents, although not a part of this test procedure, amplify or clarify its contents.

[IR1] Jelly XML processing engine, http://jakarta.apache.org/commons/jelly/.

[IR2] Eclipse http://www.eclipse.org/.

[IR3] SOAP Simple Object Access Protocol 1.1, W3C Note 08 May 2000, http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

[IR4] Hypertext Transfer Protocol -- HTTP/1.1, RFC 2616, U.C. Irvine, DEC W3C/MIT, DEC, W3C/MIT, W3C/MIT, January 1997, http://www.normos.org/ietf/rfc/rfc2616.txt

[IR5] Web Services Description Language (WSDL) 1.1, W3C Note 15 March 2001,http://www.w3.org/TR/wsdl

[IR6] XML Schema, http://www.w3.org/TR/xmlschema-0/, W3C Recommendation, 2 May 2001.

[IR7] Extensible Mark-up Language (XML) 1.0, W3C Recommendation 10 February 1998, http://www.w3.org/TR/REC-xml.

[IR8] XSL Transformations (XSLT) Version 1.0, W3C Recommendation 16 November 1999, http://www.w3.org/TR/xslt.

[IR9] Web Services Addressing (WS-Addressing) http://www.w3.org/Submission/ws-addressing/

[IR10] Software — Part 2: Document requirements definitions (DRDs), ECSS--E--40 Part 2B, 31 March 2005.

2. SYSTEM DESIGN OVERVIEW

The Toolbox software is subdivided into two separate applications:

- Toolbox Runtime Environment
- Toolbox Development Environment

The first is a configurable application that helps the Service Provider to easily convert its service in SOAP [IR3] based service without code development.
Using the TOOLBOX Runtime Environment the new Web Services can be integrated within the SSE framework. The TOOLBOX Runtime Environment allows the possibility of deploying a SOAP interface not compliant with the SSE ICD or the EOLI-XML ICD. Thus it can also be employed outside the context of the SSE framework.

The TOOLBOX Development Environment is an IDE that allows the user develop and test a Toolbox Service.
It supports the following functionalities:

- Interface repository
- TOOLBOX service creation and management
- Service operation creation
- Script development
- Import of external resources
- Service deploy on an external TOOLBOX Runtime Environment
- Service operation testing
- Service operation debugging
- Check if more recent versions of the environment are available

2.1. Software static architecture

2.1.1. TOOLBOX run-time environment

2.1.1.1. SSE – TOOLBOX – BACK-END SERVICES COMMUNICATION

This section provides an overview of the TOOLBOX run-time environment, focusing on how it is used to integrate services into the SSE framework.
The SSE allows a service provider to go to the SSE Portal and enter the details of its service interactively into the SSE database. This information, which includes the filename of the WSDL file and the parameters to define a service request, is sufficient for the SSE system and the embedded workflow tool to connect to that service over the Internet, without any development on the SSE Portal.
In case a service doesn't support the SOAP communication mode, the Service Provider can install the TOOLBOX run-time environment and use it as a gateway between the SSE and the service itself.
On the back side, the TOOLBOX supports several kind of modes to communicate with back-end services:

- File exchange
- FTP
- HTTP

- Java API call
- Java code
- Command invocation
- Database using JDBC 2.0
- SOAP

On the front side, the TOOLBOX supports both the synchronous as well the asynchronous communication mechanism described in the SSE ICD:

- Synchronous: the SSE Portal is a SOAP client and the TOOLBOX acts only as a SOAP server. It only responds to SOAP requests. The XML response is sent back to the SSE Portal during the same HTTP exchange.
- Asynchronous: the SSE Portal is a SOAP client as well as a SOAP server. The service provider sends a SOAP message back to SSE when a result is ready. Thus the TOOLBOX implements a SOAP server as well as a SOAP client. The TOOLBOX uses the WS-Addressing protocol [IR9].

Commento [MF2]: rimuovere, sembra senza connessione col resto del documento

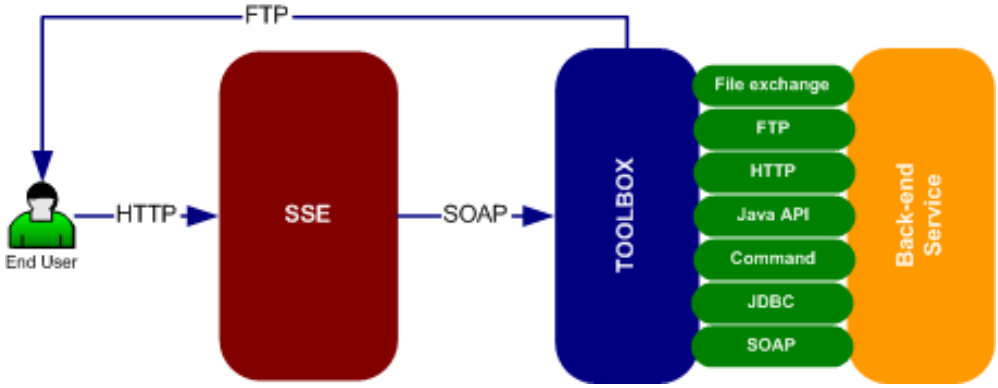

Figure 1 The SSE - TOOLBOX - Back-end service interaction

The TOOLBOX RE provides a mechanism to define the sequence of steps that are needed to complete the service each time a request arrives from the SSE. The approach used is based on XML scripting. Indeed the services deployed on the TOOLBOX are defined through a set of XML scripting files interpreted by an XML engine embedded in the TOOLBOX RE.
As seen above, a service, to be plugged in the SSE, must implement the operations defined in one of the supported interface (SSE,EOLI,HMA and OGC catalogues). The same service can support more than one operation. For each supported operation the service provider has to define a set of XML scripting files. These files define the steps required to fulfil a request incoming from the SSE Portal.

The TOOLBOX uses XML Schemas to validate all the input and the output messages exchanged with the SSE Portal. Thus the user shall provide these schemas also and redefine them when necessary (E.g. SSE schemas).

The following pictures show the communication SSE Portal-TOOLBOX-back-end service within a service request in the synchronous and asynchronous cases.

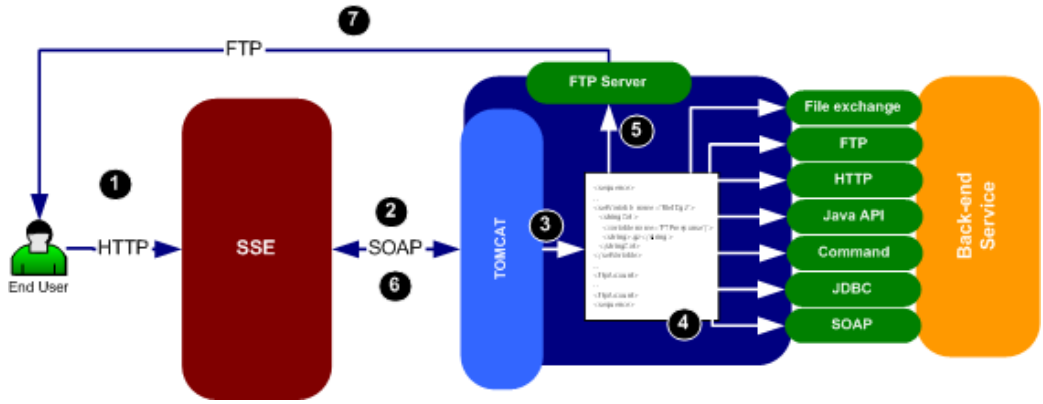

Figure 2 Synchronous communication

Synchronous message handling

1. The end user connects to the SSE portal and selects one of the deployed services.
2. Then the user asks for an operation. This causes a SOAP message be sent to the TOOLBOX.
3. The TOOLBOX receives the message from the SSE Portal. Based on the URL it addresses the request to one of the deployed services (the one corresponding to the service deployed on the SSE Portal). Based on the HTTP header “SOAPAction”, the service identifies the operation to invoke and executes the corresponding XML script.
4. The TOOLBOX starts the processing of the request interpreting the tags of the XML script. Some of these tags implement the communication with the back-end service.
5. If needed, one or more products (prepared by the back-end service) are stored on the FTP server embedded in the TOOLBOX and an FTP account is created.
6. The response to the SSE server is sent back within the same HTTP exchange. The FTP account parameters are contained in the response.
7. The User access to the service result on the portal and using the FTP account parameters contained in the response (if provided), the user can download the products.

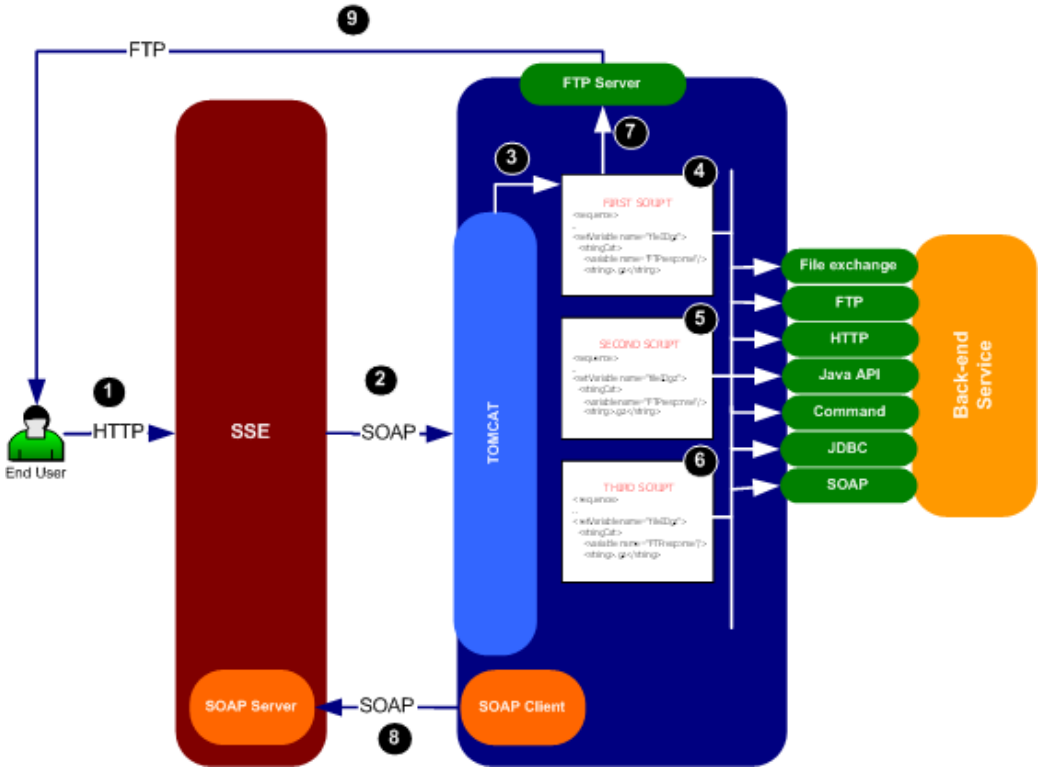

Figure 3 Asynchronous communication

Asynchronous message handling

1. The end user connects to the SSE portal and selects one of the deployed services.
2. Then the user asks for an operation. This causes a SOAP message be sent to the TOOLBOX.
3. The TOOLBOX receives the message from the SSE Portal. Based on the URL it addresses the request to one of the deployed services (the one corresponding to the service deployed on the SSE Portal). Based on the HTTP header “SOAPAction”, the service identifies the operation to invoke and executes the corresponding First XML script.
4. The TOOLBOX starts the processing of the request interpreting the tags of the First script (typically used to start the back-end service).
5. Once the First Script has been executed, the Second script is executed until a certain condition is verified (e.g. the requested product is ready).
6. Once the Second Script has returned true, the Third script is executed.
7. In the three scripts, one or more ordered products can be stored on the FTP server embedded on the TOOLBOX and an FTP account is created. The account parameters are put in the response to be pushed to the SSE server.

8. The SOAP client embedded in the TOOLBOX sends back the response to the SSE server.
9. The User access to the service result on the portal and using the FTP account parameters contained in the response (if provided), the user can download the products.

2.1.1.2. BACK-END COMMUNICATION MECHANISM

In the following section a description of the communication mechanisms supported by the TOOLBOX to deploy the Service Provider backend system is reported. For each section a template service behaviour is described.

File exchange

The communication between the TOOLBOX and the Legacy System is based on file exchange on local or network file system. The Service Provider back-end waits for a XML or text file (formatted according to a proprietary or standard format) containing a request, polling a certain location of the local disk resource. This procedure can be manual or automatic: the backend software or the operator reads the input file and initiates the requested operations, generates a response file and stores it on the file system. Thus the TOOLBOX RE shall act as a bridge between SOAP and a file exchange mechanism.
All the information needed to implement this communication paradigm (naming conventions, directory structure, etc.) shall be provided by the Service Provider using one (synchronous) or three (asynchronous) XML files. The management of these files shall be handled by the management Web Application included in the TOOLBOX.

FTP (PUSH and PULL support)

The communication between TOOLBOX and the Legacy System is based on file exchange on FTP. A (one way) communication channel between TOOLBOX and the Service Provider Legacy System has a source and a destination. These two communication actors must agree on which communication mode to use among Push and Pull.

Push Mode

In Push mode the source acts as a FTP client, making available exchanged files on a FTP server on destination side. The destination application has to perform polling on the local disk resource in order to really receive and process the received file.

Pull Mode

In Pull mode the source acts as a FTP server on which the exchanged file must be available. In order to really receive and process the exchanged file the destination application has to perform FTP polling (FTP client).
All the information needed to implement this communication paradigm (naming convention, FTP addresses, user name, password, etc.) shall be provided by the Service Provider using one (synchronous) or three (asynchronous) XML files. The management of these files shall be handled by the management Web Application included in the TOOLBOX RE.

HTTP

This scenario is applicable when the provided Service is already a Web Application. In this case the TOOLBOX RE can be configured to act as an HTTP client. The incoming SOAP message will be converted into an HTTP request to be sent to the Service Provider Web server. The HTTP response will be handled and converted into a SOAP response to be sent to the client. All the information needed to implement this communication paradigm (HTTP addresses, etc.) shall be provided by the Service Provider using one (synchronous) or three (asynchronous) XML files. The management of these files shall be handled by the management Web Application included in the TOOLBOX RE.

Java API call

This type of backend communication is applicable when the TOOLBOX RE is used to deploy an already working service implemented in Java. This could happen if, for example, the Service Provider has already developed a Web Application based on Servlets. In this scenario the integration of the service could be done re-using the legacy Java code inside the TOOLBOX RE. The reflection properties of Java language allow such a dynamic link between separately developed chunks of Java code. The TOOLBOX RE offers great flexibility, allowing virtually any interface exposed by legacy classes, to be used. The price of such a freedom is the need to tell the TOOLBOX RE how to map the XML payload (exchanged with SSE), on the Java Data Structures necessary to perform the proper calls to the legacy classes.

Java code

In this version of TOOLBOX RE it shall be possible to declare and execute a snippet of Java code directly into the XML scripting language. This kind of backend communication is an enrichment of the already existing Java API call, allowing the final user connect its service scripts to java code without the need of creating a Java class. TOOLBOX REwill automatically create the class and invoke its fields, starting from the provided Java code.
The Java code will be placed directly into the TOOLBOX script, as a CDATA section of the new java tag.
The following snippet clarifies this concept:

```
<java>
        <imports>
        import java.text.SimpleDateFormat;
        import java.util.Date;
</imports>
<code><![CDATA[
        String dateStr=null;
        SimpleDateFormat formatUtil=new SimpleDateFormat("yyMMddHHmmss");
        Date now=new Date();

        dateStr=formatUtil.format(now).toString();
        return dateStr;
    ]]></code>
</java>
```

In order to successfully run a Java snippet of code the user shall provide the full list of imports and the code to be executed. These two elements will be used to create a class that will be compiled and executed. The Java code provided shall not contain any class or method definition.
The TOOLBOX RE engine will take care of adding extra Java code that creates a copy of the engine's variables in order to let the user interact with them inside the code snippet. At the end of the code, all variables defined before the <java> tag will be copied back to the TOOLBOX RE engine and will be accessible by the rest of the tags. All variables created by the code snippet will have its scope limited to the tag <java> and they won't be copied to the TOOLBOX RE engine at the end of the tag execution.
The following clarifies this concept:

```
<setVariable name="dateFormat">
<string>yyMMddHHmmss</string>
</setVariable>
<java>
        <imports>
        import java.text.SimpleDateFormat;
        import java.util.Date;
</imports>
<code><![CDATA[
        String dateStr=null;
        SimpleDateFormat formatUtil=new SimpleDateFormat(dateFormat);
        Date now=new Date();

        dateStr=formatUtil.format(now).toString();
        return dateStr;
    ]]></code>
</java>
```

The variable dateFormat, created into the TOOLBOX RE engine through the setVariable tag, is made available to the Java code. If the variable is modified by the Java code, its content is copied into the TOOLBOX RE engine. The variable dateStr will be destroyed at the end of the <java> tag execution.
This mechanism is limited to a set of types, as listed below:

- java.lang.Byte
- java.lang.Short
- java.lang.Integer
- java.lang.Long
- java.lang.Float
- java.lang.Double
- java.lang.Character
- java.lang.Boolean
- java.lang.String
- org.w3c.dom.Document

Script support

This type of backend communication is applicable when the TOOLBOX is used to deploy services implemented using a scripting mechanism. In this scenario the integration of the service could be done using script calls inside the TOOLBOX. The mapping between the SOAP message and the script call to be performed will be provided by the Service Provider using one (synchronous) or three (asynchronous) XML files (see examples in 2.6.3). The management of these files shall be handled by the management Web Application included in the TOOLBOX.

JDBC DBMS support

This type of backend communication is applicable when the TOOLBOX RE is used to deploy a service based on an existing DBMS. In order to apply this integration method a JDBC driver for the DBMS must be available. The JDBC driver allows the DBMS to be accessed by any Java application, through standard SQL queries, getting back standard Result Set structures. The TOOLBOX RE will use the incoming data of the SOAP call to perform a set of DB accesses, and use the obtained result to build the SOAP response. The Service Provider using (synchronous) or three (asynchronous) XML files will provide the mapping between the SOAP message and the query to be performed. The management of these files shall be handled by the management Web Application included in the TOOLBOX RE.

SOAP

This type of backend communication is applicable when the TOOLBOX RE is used to deploy a service having already a SOAP interface. In this case the TOOLBOX RE allows the service to expose a different SOAP interface. A meaningful example is when the service has to be integrated in SSE but its SOAP interface is not compliant with the SSE ICD **Errore. L'origine riferimento non è stata trovata.**. The service can exchange RPC-oriented or document-oriented messages. Communication with SSE occurs in the following way: the SOAP message incoming from SSE is converted into a new SOAP request to be sent to the Service Provider SOAP server. The request body contains a document compliant with the service provider SOAP interface. The SOAP response is analysed and converted by the TOOLBOX RE (e.g using the xsl transformation mechanism) into a SOAP response compliant with the SSE ICD before being sent back to the SSE system. Also communication on the Secure Socket Layer is supported.

2.1.1.3. Catalogues support

Within this project the SOAP layer will be used to handle a new service type: the catalogue.
Two approaches are envisaged for the publication of a catalogue within the Toolbox: The Proxy and the Stand-Alone approaches.

Proxy approach

If Collections or Earth Observation metadata are already stored in legacy (non OGC-compliant) catalogues,the user will be able to provide an OGC compliant interface onto existing data by means of the SSE Toolbox.

The incoming requests will be linked on the fly to the legacy catalogue, and result sets will be converted from the legacy format to the OGC CSW ebRIM format and responses sent back to the user.
Some operations will be directly implemented on the Toolbox and will be partly handled via configuration files (e.g. getCapabilities, desribeRecord) other interfaces will be directly mapped to operations (one or more operations handled within the Toolbox to collect and merge metadata) available on the legacy catalogue. The connection with the Legacy database can be implemented either via Java code or using the scripting language provided by the Toolbox (e.g. via the JDBC tags, via HTTP and/or SOAP and XSL transformations). In both cases the user will have to provide the logic to convert legacy metadata into OGC compliant metadata. The complexity of such implementation depends on the legacy catalogue structure.

Stand-Alone approach

In this approach a built in catalogue compliant with the HMA interfaces is provided by the Toolbox. The SOAP layer is directly connected with the catalogue layer and the user does not have to provide the integration logic. The catalogue included in the Toolbox is used as a repository (for storing data) and a registry (for indexing data). The data have to be provided and ingested in the system by the user.
After a first analysis the OMAR open source software has been selected for the integration. It is OGC compliant and as result of this project it will provide all capabilities and operations defined in OGC ebRIM Application Profile for CSW document. It will be able to map incoming EO Products to objects defined by the Data Model described in the specification.
Every Earth Observation Product will be stored within the Catalogue and indexed to allow complex queries and fast retrieving. The OMAR software will be integrated connecting directly the SSE Toolbox to the core component implementing the EbRim functionalities. Thus the SOAP layer implemented in the Toolbox will be maintained. The message extracted will then be handled directly by the OMAR core components and the XML result message will be returned back to the Toolbox. This approach ensures that all the layers that will be implemented on the Toolbox (e.g. security) will be available in the future for the catalogue functionalities.
Moreover the testing and monitoring Toolbox functionalities will be available also for the EbRim catalogue services.

2.1.1.4. RETRY ON THE ASYNCHRONOUS EXCHANGES

An operation deployed in asynchronous mode can be configured such that if pushing the response to SSE fails, a number of new attempts can be performed. The number of attempts and the rate every attempt occurs at are configurable.

2.1.1.5. PERSISTENCE OF VARIABLES AMONG ASYNCHRONOUS OPERATION SCRIPTS

For operations deployed in asynchronous mode, the TOOLBOX RE supports persistence of some types of variable among the executions of the three scripts describing the operation. This means that a variable defined in the First script can be referred, e.g., in the third script.
Persistence concerns the following variable types:

- byte (whose value is an integer number comprised between -27 and 27-1)
- short (whose value is an integer number comprised between - 215 and 215-1)
- integer (whose value is an integer number comprised between – 231 and 231-1)
- long (whose value is an integer number comprised between – 263 and 263-1)
- float (largest positive finite value: (2-2-23)•2127, smallest positive nonzero value: 2-149)
- double (largest positive finite value: (2-2-52)•21023, smallest positive nonzero value: 2-1074)
- char (16-bit Unicode character)
- boolean
- string
- DOM Document

2.1.1.6. CLEANUP MECHANISM

The TOOLBOX RE supports a cleanup mechanism allowing the service provider to specify the wished status in case of failure of a script.
This mechanism foresees the introduction of the following tags:
· <cleanupProcedure>
It is a sequence of </cleanup> tags:

```
<cleanupProcedure>
        <cleanup [marker="Marker_One"]>
                .
                .
        </cleanup>
        <cleanup [marker="Marker_Two"]>
                .
                .
        </cleanup>
        <cleanup [marker="Marker_Three"]>
                .
                .
        </cleanup>
</cleanupProcedure>
```

Any </cleanup> tag encloses an element containing the operation(s) to be executed in case of failure or request timeout. If the optional “marker” attribute is specified, the execution is subordinated to the existence of such marker (see <addCleanupMarker /> and <removeCleanupMarker />), otherwise the execution occurs in any case.
· < addCleanupMarker value="A_Marker"/>
It is used to mark a point of one of the three scripts whose achievement is required by the execution of the related <cleanup> tag. For example, let’s suppose the deletion of a file created during the execution of a script is wished in case of failure: the execution of the <cleanup> tag deleting that file must occur only if the file was created, i.e. the failure occurred after its creation.
EXAMPLE:

```
<?xml version="1.0" encoding="UTF-8"?>
<service queuing="false" serviceName="TEST" serviceSchema="mass.xsd" suspendMode="soft" wsdlTargetNS=""
xmlns="http://pisa.intecs.it/mass/toolbox/serviceDescriptor" xmlns:com="http://pisa.intecs.it/mass/toolbox/common"
xmlns:mtbs="http://pisa.intecs.it/mass/toolbox/xmlScript">
```

```
    <schemas>
        <schema schemaName="mass.xsd"/>
        <schema schemaName="AOIFeatures.xsd"/>
        <schema schemaName="feature.xsd"/>
        <schema schemaName="geometry.xsd"/>
        <schema schemaName="oi.xsd"/>
        <schema schemaName="xlinks.xsd"/>
    </schemas>
    <operations>
        <operation name="RFQ" type="synchronous">
            <script soapAction="test">
                <mtbs:sequence xmlns:mtbs="http://pisa.intecs.it/mass/toolbox/xmlScript">
                    <mtbs:setVariable name="fileName">
                        <mtbs:string>/home/toolbox/request.xml</mtbs:string>
                    </mtbs:setVariable>
                    <mtbs:dumpXML>
                        <mtbs:variable name="fileName"/>
                        <mtbs:xmlRequest/>
                    </mtbs:dumpXML>
                    <mtbs:addCleanupMarker value="A_Marker"/>
            .
            .
            .
                    <mtbs:cleanupProcedure>
                        <mtbs:cleanup marker="A_Marker">
                            <mtbs:fileDelete>
                                <mtbs:variable name="fileName"/>
                            </mtbs:fileDelete>
                        </mtbs:cleanup>
                        <mtbs:cleanup marker="Another_Marker">
                        .
                        .
                        </mtbs:cleanup>
                    </mtbs:cleanupProcedure>
                </mtbs:sequence>
            </script>
        </operation>
    </operations>
</service>
```

Existence of the “A_Marker” marker ensures existence of the file named “fileName”.

- < removeCleanupMarker value="A_Marker"/>

This tag removes a tag previously added. It is used to mark a point of one of the three scripts (First, SecondSecondsecond, Third script) whose achievement removes the need to execute the cleanup tag(s) associated with that marker.
EXAMPLE:

```
<?xml version="1.0" encoding="UTF-8"?>
<service queuing="false" serviceName="TEST" serviceSchema="mass.xsd" suspendMode="soft" wsdlTargetNS=""
xmlns="http://pisa.intecs.it/mass/toolbox/serviceDescriptor" xmlns:com="http://pisa.intecs.it/mass/toolbox/common"
xmlns:mtbs="http://pisa.intecs.it/mass/toolbox/xmlScript">
    <schemas>
        <schema schemaName="mass.xsd"/>
        <schema schemaName="AOIFeatures.xsd"/>
        <schema schemaName="feature.xsd"/>
        <schema schemaName="geometry.xsd"/>
        <schema schemaName="oi.xsd"/>
        <schema schemaName="xlinks.xsd"/>
    </schemas>
    <operations>
        <operation name="RFQ" type="synchronous">
            <script soapAction="test">
                <mtbs:sequence xmlns:mtbs="http://pisa.intecs.it/mass/toolbox/xmlScript">
```

```
                .
                        <setVariable name="fileName">
                            <string>/home/toolbox/request.xml</string>
                        </setVariable>
                        <dumpXML>
                            <variable name="fileName"/>
                            <xmlRequest/>
                        </dumpXML>
                        <addCleanupMarker value="A _Marker"/>
                .
                .
                .
                        <fileDelete>
                            <variable name="fileName"/>
                        </fileDelete>
                        <removeCleanupMarker value="A_Marker"/>
                .
                .
                .
                        <cleanupProcedure>
                            <cleanup marker="A _Marker">
                                <fileDelete>
                                    <variable name="fileName"/>
                                </fileDelete>
                            </cleanup>
                            <cleanup marker="Another_Marker">
                            .
                            .
                            </cleanup>
                        </cleanupProcedure>
                    </mtbs:sequence>
                </script>
            </operation>
        </operations>
    </service>
```

In this case the First script itself foresees the deletion of the file named “fileName”: if the failure occurs after the deletion the cleanup tag marked with “A_Marker” must not be executed.
NOTE: A marker created in a script can be used also in the other scripts as shown in the following example.
EXAMPLE:

```
<?xml version="1.0" encoding="UTF-8"?>
<service queuing="false" serviceName="vv" serviceSchema="mass.xsd" suspendMode="soft" wsdlTargetNS=""
xmlns="http://pisa.intecs.it/mass/toolbox/serviceDescriptor" xmlns:com="http://pisa.intecs.it/mass/toolbox/common"
xmlns:mtbs="http://pisa.intecs.it/mass/toolbox/xmlScript">
    <schemas>
        <schema schemaName="mass.xsd"/>
        <schema schemaName="AOIFeatures.xsd"/>
        <schema schemaName="eoli.xsd"/>
        <schema schemaName="feature.xsd"/>
        <schema schemaName="geometry.xsd"/>
        <schema schemaName="oi.xsd"/>
        <schema schemaName="xlinks.xsd"/>
    </schemas>
    <operations>
        <operation name="Order" pollingRate="5s" requestTimeout="30m" retryAttempts="2" retryRate="2m"
type="asynchronous">
            <admittedHosts>
                <admittedHost>www.host.name</admittedHost>
                <admittedHost>www.host2.name</admittedHost>
            </admittedHosts>
            <script soapAction="sendOrder">
                <mtbs:sequence xmlns:mtbs="http://pisa.intecs.it/mass/toolbox/xmlScript">
```

```
                <mtbs:setVariable name="fileName">
                    <mtbs:string>/home/toolbox/request.xml</mtbs:string>
                </mtbs:setVariable>
                <mtbs:dumpXML>
                    <mtbs:variable name="fileName"/>
                    <mtbs:xmlRequest/>
                </mtbs:dumpXML>
                <mtbs:addCleanupMarker value="A _Marker"/>
            .
            .
            .
                <mtbs:cleanupProcedure>
                    <mtbs:cleanup marker="A _Marker">
                        <mtbs:fileDelete>
                            <mtbs:variable name="fileName"/>
                        </mtbs:fileDelete>
                    </mtbs:cleanup>
                    <mtbs:cleanup marker="Another_Marker">
                    .
                    .
                    </mtbs:cleanup>
                </mtbs:cleanupProcedure>
            </mtbs:sequence>
        </script>
        <script>
            <mtbs:sequence xmlns:mtbs="http://pisa.intecs.it/mass/toolbox/xmlScript">
            .
            <cleanupProcedure>
                    <cleanup marker="A _Marker">
                        <fileDelete>
                            <variable name="fileName"/>
                        </fileDelete>
                    </cleanup>
                </cleanupProcedure>
            .
            </mtbs:sequence>
        </script>
        <script>
            <mtbs:sequence xmlns:mtbs="http://pisa.intecs.it/mass/toolbox/xmlScript">
            .
            .
            .
            </mtbs:sequence>
        </script>
    </operation>
  </operations>
</service>
```

Another important remark is that the service provider can put the <cleanupProcedure> tag in any position within the script, without being worried about what the script must return.

2.1.1.7. HUMAN TASK

In this version of TOOLBOX RE it shall be possible to synchronize service instances with human tasks. The synchronization will be realized through a set of XML tag of the scripting language and some administration pages.
Through new XML tags it shall be possible to:

- Add a synchronization point
- Check the instance status
- Retrieve the instance error description (if any)

There won't be any restrictions over the usage of these new tags, so it will be possible to provide this synchronization in any kind of operation.

Through the administration web interface it shall be possible to display all the orders that belong to a service and that are still in pending status. Through this interface it shall be possible to select one or more blocked instances and unlock them. If the human task procedure fails, it shall be possible to provide an error status.
This new feature can be used for payments confirmation but won't help in any way to realize a payment procedure: it will only provide a synchronization point for this specific kind of human task. Payments have to be handled outside the SSE framework.

2.1.1.8. SUPPORT FOR THE AUTOMATIC WSDL AND WSIL FILES GENERATION AND PUBLISHING VIA THE TOOLBOX

When a new service is deployed all information on its SOAP interface is published on the TOOLBOX installation host by means a WSDL file automatically created. The WSDL file will have the structure specified in the SSE ICD, even for services whose SOAP interfaces will not support the SSE ICD.
The TOOLBOX RE builds a WSDL starting from the template contained in the XSL stored in <TOMCAT_ROOT>/webapps/TOOLBOX/WEB-INF/XSL/emptyWSDL.xsl.
The TOOLBOX completes the XML returned by the XSL transformation (applied to an "empty" XML) with the information filled by the service provider in the TOOLBOX RE ADMINISTRATOR Web Application and according to the operations the service exposes.
The values of the three parameters ("serviceName", "targetNS", "url") contained in the XSL above are changed at transformation time:

- "serviceName": the name of the service as deployed on the TOOLBOX.
- "targetNS": the service WSDL target namespace.
- "url": the HTTP URL a service is deployed at (i.e., the address the SOAP requests must be sent to).

Moreover the new service is added to the WSIL file. Web Services Inspection Language (WS-Inspection)is a service discovery mechanism designed around an XML-based model for building an aggregation of references to existing Web service descriptions, which are exposed using standard Web server technology.

2.1.1.9. TOOLBOX CONFIGURATION

The configuration of the TOOLBOX RE is performed using a Web Application (TOOLBOX Administrator), composed by some JSP pages plus several classes invoked by the JSP pages themselves. Configuration can partially occur using the TOOLBOX development environment via SOAP.
The TOOLBOX RE Administrator manages the list of the currently deployed services and the basic mechanisms of every service operation: creation, SOAP deployment, configuration, starting, monitoring and deletion. The Web Application supports five main categories of tasks:

TOOLBOX configuration and administration	Concerning all the general configuration issues regarding the TOOLBOX RE Web Application, such as log directory, log level, Tomcat installation information, use of Apache as Proxy, etc.
Service configuration	Concerning all the issues supported and allows passing the

and Deploying	TOOLBOX RE all the necessary information in order to integrate an existing service in the SSE infrastructure. More in general, this category of tasks includes all the steps required to "build" a SOAP interface for an existing service (back-end) using the TOOLBOX RE. Another task included in this category is the importing end exporting of services previously deployed on the TOOLBOX DE
Service monitoring	Concerning monitoring of the TOOLBOX RE status and exchanged data while the service is working. It manages things like log files, pending requests, etc., letting the administrator act on those items using a browser instead of doing the hard job on actual files and directories. Examples of such operations are: • Viewing the status of the Service instances • Cancelling pending instances • Viewing the service activity overview • Viewing the instance flow • Managing Log files
Service testing	Concerning functionalities to test services and catalogues without having to deploy them on the SSE Portal. Services and Catalogues to test can be deployed on the TOOLBOX RE or have their own SOAP interface. Whilst the catalogue test page builds a SOAP message compliant with the EOLI-XML ICD, the generic service test page can be used to send any SOAP message.
FTP server user management	Concerning creation and deletion of users in the FTP server embedded in the TOOLBOX RE

2.1.1.10. REMOTE LOG ACCESS

The Toolbox RE allows accessing the log information via URL for a given Order ID and publishes the service log as a RSS feed.
The URL has the following structure:

- Asynchronous requests: http://<TOOLBOX_HOST>:<TOOLBOX_PORT>/TOOLBOX/viewAllServicesInstances.jsp?orderId=<ORDER_ID>&pushHost=<PUSH_HOST>
  Example:http://toolbox.pisa.intecs.it/TOOLBOX/viewAllServicesInstances.jsp?orderId=AAAAAOOOOOSSSS&pushHost=213.173.101.148
- Synchronous requests :http://<TOOLBOX_HOST>:<TOOLBOX_PORT>/TOOLBOX/viewAllServicesInstances.jsp?orderId=<ORDER_ID>
  Example:http://toolbox.pisa.intecs.it/TOOLBOX/viewAllServicesInstances.jsp?orderId=BB808080

The Toolbox RE will display a page with access to all instances associated to that key. The authentication page is displayed if the user is not authenticated yet. After the login page the user is directed to the requested page (log instance page).

Moreover it is possible to access log information via URL for a given service. The URL has the following structure:

http://<TOOLBOX_HOST>:<TOOLBOX_PORT>/TOOLBOX/viewServiceLog.jsp?serviceName=<SERVICE_NAME>

The Toolbox RE will display the service log page. The authentication page is displayed if the user is not authenticated yet. After the login page the user is directed to the requested page (service log page).

For each service a specific RSS feed file is created in order to keep track of all activities without the need of logging into the Toolbox administration web pages.
The RSS feed link can be found into the administration web pages, associated to the icon below:

When a service is selected from the combo box, the icon is shown on the service specific menu bar. The link can be copied through the browser provided functionalities (e.g. right click and select “Copy link location”).
Any RSS feed reader compliant with RSS version 2.0 can be used.

2.1.2. TOOLBOX development environment.

2.1.2.1. INTERFACE REPOSITORY

The TOOLBOX Development Environment will implement an interface repository that will allow the user to define a set of interfaces that will be used in multiple services. This will speed up and ease the development of TOOLBOX service, reducing the effort for the service creation. Using the interface repository the development will be more focus on the script development than on the service structure creation.
The interface repository will host both interfaces defined by the user (through Eclipse wizards) and predefined interfaces available out of the box installing the TOOLBOX Development Environment plugin. The final user will have SSE, EOLI,HMA ordering and HMA OGC catalogues interfaces already available after the TOOLBOX Development Environment installation. Moreover using the Eclipse Update functionality, the interfaces set can be updated (in case of errors) or populated with additional interfaces supported by the SSE and HMA.

2.1.2.2. TOOLBOX SERVICE CREATION AND MANAGEMENT

The TOOLBOX Development Environment will implement a set of wizards that let the user create a new TOOLBOX service from scratch. All service related information (like service schemas, queuing information, suspend mode, etc) are collected by the wizard and stored on a new Eclipse project. These information are managed and maintained by TOOLBOX Development Environment during all the development process and transferred to the TOOLBOX Runtime Environment when the service is deployed.

A set of property page will let the final user to modify service related properties after its creation.
The service creation wizard will let the user associate an already defined interface taken from the repository or simply let the user specify a new one. In the latter case, the schema set and WSDL information are collected later during the service development.
The service wizard will also handle all service types (standard or catalogue service) and their associated modes (e.g. Stand-Alone and Gateway modes).

2.1.2.3. SERVICE OPERATION CREATION

The TOOLBOX Development Environment will implement a new set of wizards that will let the user add new operations to an already created service.
These wizards will let the user:

- Choose from a defined list of operations when the service has been created selecting its interface from the repository
- Specify all operation related information when the service has been created without a predefined interface.

In any case the operation creation wizard will provide a set of template scripts for the new operation. These will be used as start point for the operation implementation. The script set can be enriched adding new TOOLBOX Scripts through Eclipse wizards.

2.1.2.4. SCRIPT DEVELOPMENT

The XML scripting editor supports both textual and graphical views. The tools provided allows editing an XML file both in textual and graphical format, helping the writing phase with features like syntax colouring, context sensitive help and online validation.
The editing experience, using the design view, follows a schema-driven path, where all pieces that build up the Toolbox script are selected through context menus, populated by tags taken from the Toolbox schema. Using this view, the Toolbox script is valid by construction, not allowing the user to make errors.
The text view, due to its nature, follows a more free path, where all pieces of the script are typed manually by the user and online validation is made during typing. Features like tag suggestion, validation messages and error highlighting are implemented to help the user build a valid script.
In this view, documentation taken from the schema is displayed to the user through pop up frames, showed hovering over tags.
For each edited file, both views are maintained aligned during editing, letting the user experience the most comfortable writing experience, exploiting all good features of each one.

2.1.2.5. IMPORT OF EXTERNAL RESOURCES

All kind of resources (like XSL and template files) can be added to the project using import wizards. These will be managed by the TOOLBOX Development Environment, becoming part of the service that is being developed. The deploy functionality will take care of these files too, publishing them together will all other resources (scripts, schema etc).
This will make the deploy process even easier, avoiding the need of uploading all external resources on the host where the service is being published.

2.1.2.6. SERVICE DEPLOY ON AN EXTERNAL TOOLBOX RUNTIME ENVIRONMENT

The TOOLBOX Development Environment will implement a set of functionalities that will let the user deploy/undeploy a service into a TOOLBOX Runtime Environment. These functionalities can be used to easily deploy the service on an operational or testing environment. The deploy/undeploy functionalities will be implemented as context sensitive pop up menu, available only when the service project is selected by the user on the Eclipse workspace.
In order to achieve the goal of remotely deploy a service, the TOOLBOX RE will be enriched with an HTTP interface that will ingest a zip archive containing all the files necessary for the porpouse (scripts, schemas etc). This interface will be used by the TOOLBOX DE when the user asks to deploy the service.

2.1.2.7. SERVICE OPERATION TESTING

The TOOLBOX Development Environment will be enhanced in order to let the user test its services directly on a TOOLBOX Runtime Environment. Using this functionality the developed service can be tested in any environment, at least directly on the operational one.
In order to achieve this goal, both Development and Runtime Environment will be enhanced and linked.
The TOOLBOX Development Environment will allow the user test a service's operation, defining multiple test cases (called launch configuration in Eclipse) that can be used several times. For each test case an input file shall be selected from one of those available on the service Eclipse project. New test files can be added to the project using wizards, allowing the user create a set of input data.
All test cases defined by the user will not depend on the TOOLBOX Runtime host where the service is deployed/tested so they can be used on multiple hosts without any change.
All messages exchanged between TOOLBOX Development Environment and the tested service will be stored on the Eclipse project in order to let the user analyse and store them (if needed).
The following is the list of files collected when testing a synchronous operation:

- Input file sent to the service
- Response file sent by the service when the operation execution has been completed

The following is the list of files collected when testing an asynchronous operation:

- Input file sent to the service
- Acknowledge message sent by the service
- Output file pushed by the service when the execution has been completed.

2.1.2.8. SERVICE OPERATION DEBUGGING

The testing functionalities of the Toolbox Development Environment will be enhanced with debugging functionalities also. Through these a service operation can be executed in a specific mode that allows the user:

- Setting/removing of breakpoints.

- Suspend execution when a breakpoint is hit.
- Highlight suspension point directly into the source code.
- Display the variables value (basic types and XML documents)
- Modify variables value (basic types and XML documents)
- Display the execution tree (as shown into the Toolbox Runtime Environment)
- Save input, Acknowledge and output messages
- Save execution trees for executed scripts

Breakpoints can be added and removed clicking on the source file, at the desired line. The first tag found at the line will be considered as target of the breakpoint. Each breakpoint will suspend the execution when it is hit. During suspension, all variables will be displayed to the user together with their values. For the following types the value can be modified through the GUI:

- java.lang.Byte
- java.lang.Short
- java.lang.Integer
- java.lang.Long
- java.lang.Float
- java.lang.Double
- java.lang.Character
- java.lang.Boolean
- java.lang.String
- org.w3c.dom.Document

The new value will be available to the script as soon as the execution is resumed.
During suspension the execution tree is displayed, as it is on the Toolbox administration pages. Thanks to this, all executed tags can be checked, taking a look to each calculated value and returned by tags and/or attributes. Execution trees for response builder, first, send and third scripts will also available to the user for offline visualization and inspection. A copy of these will be stored into the project together with input, acknowledge and output message. This will ease the development and debugging of errors.

2.1.2.9. TOOLBOX VERSION CHECK

To automate the upgrade procedure the TOOLBOX will check automatically if a new version is available. Such check may occur any time the user launch the TOOLBOX development plug-in or when specified by the user, as permitted by the upgrade infrastructure of Eclipse.
Thanks to the Eclipse infrastructure, the Toolbox Development Environment can be installed, upgraded and patched with few easy steps without worrying about dependencies check.
In case a new version of the plug-in exists, Eclipse notifies the user.
The user will then be able to access a page describing the new features available.

An offline procedure is also provided in order to install, upgrade and patch an installation of Eclipse where an Internet connection is not directly available. A download page has been added to the Toolbox web site on Intecs in order to accomplish this objective.

3. SOFTWARE TOP-LEVEL ARCHITECTURAL DESIGN

3.1. Overall Architecture

3.1.1. Run-time environment

The TOOLBOX high-level architecture is illustrated in Figure 4 There's only one Servlet for all the services deployed on TOMCAT. The TOOLBOX RE HTTP Servlet acts as a dispatcher:

- receives HTTP requests having SOAP requests as body;
- gets from its Service table the service the SOAP request is addressed to
- delivers the request in the form of a DOM, together with the SOAPAction extracted from the HTTP header

The Service:

- performs some general control activities, such as queuing requests, synchronizing resource access by the different threads of the TOOLBOX RE (polling thread for asynchronous operations, de-queuing threads, TOMCAT thread executing calls etc.)
- dispatches the request to the Operation the request refers to (according to the corresponding SOAP Action)

The Operation

- manages unique key of requests (orderId and the Push host) for asynchronous operations and retrieves the related script.
- in case of asynchronous operations passes the DOM representing the request to an instance of a RequestHandler which executes the proper script and performs request status management tasks.
- in case of synchronous requests the passes the DOM representing the script directly to the TOOLBOX Engine in order to be executed.

The TOOLBOX RE engine uses some Open Source tools in order to perform some standard tasks:

- Jelly libraries, XPath processing, HTTP Client and FTP server developed under the Apache License (http://www.apache.org/licenses/);
- FTP client, edtFTPj ftp client licensed under the LGPL, the GNU Lesser General Public License (http://www.enterprisedt.com/)

The TOOLBOX RE supports the persistence of the status. Thus it is possible to restart TOMCAT without losing knowledge of pending requests.
In order to restore the state of the TOOLBOX RE, an XML document is loaded from a state file as soon as TOMCAT loads it. This file stores the state of each pending request and it is updated each time the state of the TOOLBOX RE changes.
In particular the state of the application changes when:

- A new request arrives from a SOAP Client (e.g. SSE)
- A request has been de-queued for First script execution
- The First script execution is completed
- A Successful Second script has been performed
- A Third script operation has been completed
- The request timeout is expired (if it is the case)

The state of the pending requests can be displayed using the Web Application.
A service loads its state file when TOMCAT is started. Thus the Polling Thread, the Thread managing timers and the Thread managing timeout are automatically started
An administration Web application helps to manage the services deployed on a TOMCAT installation.

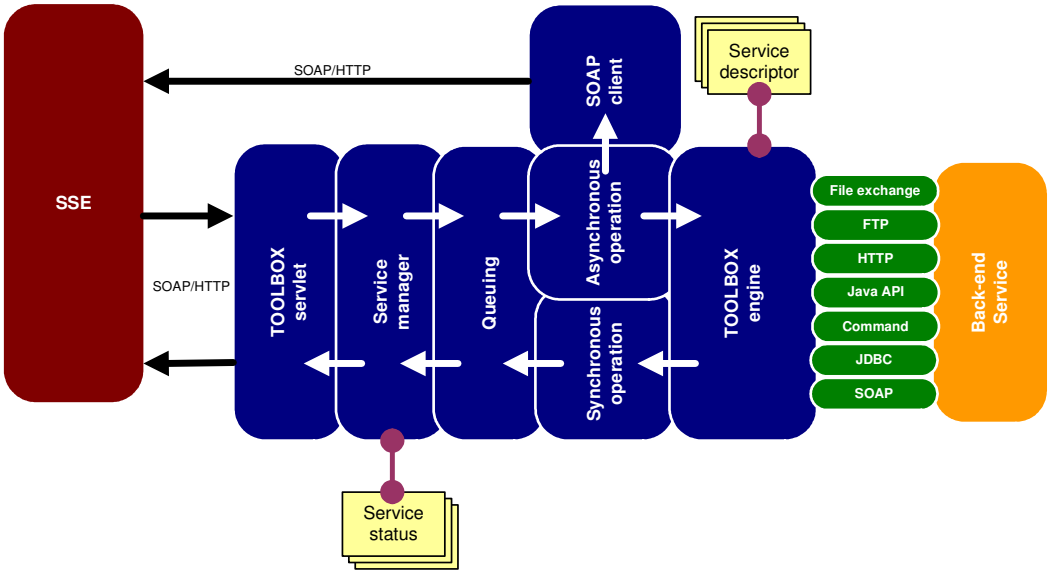

Figure 4 TOOLBOX high level decomposition

3.1.2. Development environment

In the development environment, the TOOLBOX is an Eclipse [IR2] plug-in.
It supports the following functionalities:

- Toolbox Service editor: the plug-in allows building a TOOLBOX.
- Launcher: the plug-in allows executing a TOOLBOX operation.
- Check if more recent versions are available on the TOOLBOX server.

The TOOLBOX DE plug-in is structured in multiple separate plug-ins:

- com.intecs.ToolboxScript: it contains the code which is independent from the graphical representation.
- com.intecs.ToolboxScript.ui: it contains the user interface dependent code.
- com.intecs.ToolboxScript.editorFiles: it contains all working files like Toolbox script schema, XSLs and on line documentation.
- com.intecs.ToolboxScript.libs: it contains all libraries necessary to execute the engine.
- com.Intecs.ToolboxScript.interfaces: it contains all files necessary to build the interface library

Figure 5 shows the high level dependencies between the TOOLBOX plug-in and Eclipse using the UML2 notation; please note that for the Eclipse component only the main sub-

components are represented. A distinction between core code and Toolbox plugin code has been made in order to categorize all pieces of code.

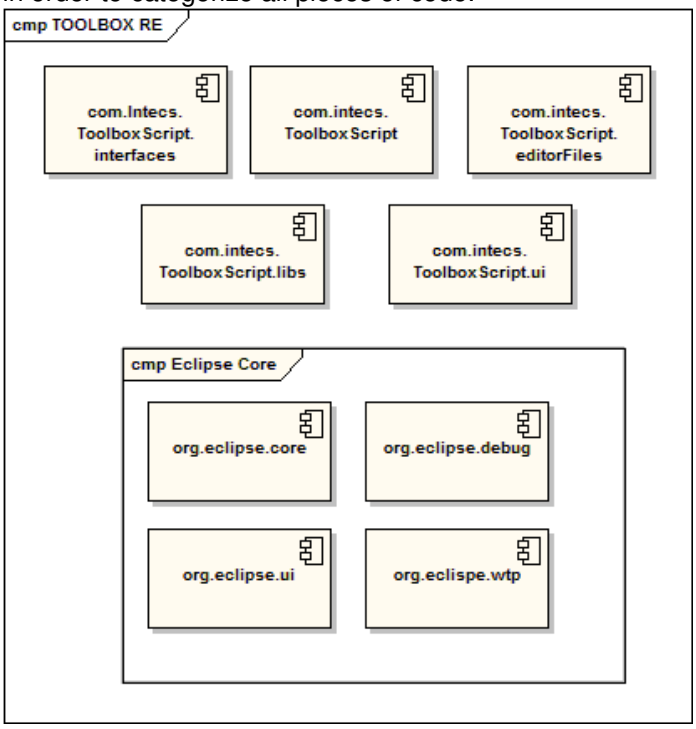

Figure 5 High level TOOLBOX DE plug-ins structure

4. Software architectural design

4.1. Software Item components

4.1.1. Toolbox

4.1.1.1. Type

The Toolbox class extends the javax.servlet.http.HttpServlet class and overrides the both the do Get and the doPost methods.

4.1.1.2. Purpose

The purpose of this class is to receive and handle all requests coming via HTTP. Several kind of requests may be sent to this servlet, obtaining different kind of behaviours.
The following are the requests kinds:

- Service requests: a message is received and I forwarded to the service in order to perform the associated task.
- Command requests: HTTP messages (POST and/or GET) can be received in order to perform several basic tasks like delete a service, retrieve the service and Toolbox logs etc..

4.1.1.3. Function

The doPost method manages the marshalling and un-marshalling of DOM Document representing SOAPMessages.
This object performs essentially the following tasks:

- On initialisation (executed only once before the first request is served) it loads its configuration containing, among other details, the list of the deployed services. Thus the TOOLBOX will contain a table of Service objects (see below) loaded during initialization.
- While running, the Toolbox receives two kinds of calls:
  - SOAP over HTTP requests directed to the doPost method
  - administration and monitoring calls from the JSPs of the WEB application associated to the TOOLBOX.
  - In both cases, the request is redirected to the selected service. Service objects have a method called processRequest. This method exchanges DOM Documents. The Toolbox transforms SOAP-HTTP requests into DOM Documents and DOM Documents into SOAP-HTTP responses. The administration and monitoring operation are simply redirected to the proper method of the Service object if necessary.

4.1.1.4. Interfaces

Being a servlet, this class exposes all interfaces needed by specification. In particular the class exposes the Java interface javax.servlet.http.HttpServlet.

4.1.2. Service

4.1.2.1. Type

The element is developed as Java class.

4.1.2.2. Purpose

The Service class implements a single service deployed on the TOOLBOX and it is essentially a collection of Operation objects (that actually fulfil the requests), contained in an “operations” table.

4.1.2.3. Function

Service objects can have a queuing mechanism for asynchronous operations. The following applies if the service requests are queued. This mechanism prevents concurrent send XML scripts to be executed. Before invoking the ToolboxEngine executeScript method the Threads executing the send XML scripts have to check if no one is executing and, if not so, they enqueue their request information and block. The blocked threads are notified every time a successful check is executed. Each blocked thread controls the queue to check if they can execute (their request information is the first item in the queue). The queuing mechanism can be at different levels:

- TOOLBOX level: all the Service objects share the same queue
- Service level: each Service has its own queue

A service can be suspended and resumed.
The Service suspension can have two modes which are selected during the service configuration procedure:

- HARD mode: the First script operations are not executed, the request is put in rejected status and the get result will be information of such rejection.
- SOFT mode: requests are queued on a dedicated queue before executing First scripts and before and after the normal queuing mechanism. Each request in suspend queue is served when the service is resumed.

A Service can also be stopped and started. These two operations (like suspend and resume) can be made from the administration WEB application. Starting a service simply means that it is ready to do its initialisation and process incoming requests, whilst stopping means to bring the service in a "blank" status. The blank status is achieved stopping each running thread (timeout, push, timers) and removing every request from the status. This allows reconfiguring the Service and then restarting it. When a request directed to a stopped Service is received by the TOOLBOX, an error message compliant with the SSE ICD is returned if the operation requested is supported by SSE, otherwise a SOAP Fault is returned.

4.1.2.4. Resources

Any service has its own configuration parameters. Moreover its status (pending requests and timers) is always saved on disk so that persistence is achieved between different runs of TOMCAT. This information is loaded when TOMCAT is started. The initialisation consists in the loading of the entire given configuration parameters and other tasks.
From the service descriptor file the “operations” table is filled with Operation objects. The key of the table is the operation name as it appears in the service descriptor file. The operations have their configuration parameters that are maintained in the service descriptor file

4.1.3. Operation

4.1.3.1. Type

The Operation is an abstract class whose code is used by two sub classes : AsynchronousOperation and SynchronousOperation.

4.1.3.2. Purpose

The purpose of this class and its derivation is to handle an incoming request according to the operation mode (synchronous or asynchronous)

4.1.3.3. Function

The **Operation class** has a “processRequest” (which actually fulfils the request) method called by the Service proceesRequest.
When the AsynchronousOperation object is initialised (upon the first call to the related SOAP service), a Polling Thread is started. This thread performs an infinite loop in which it retrieves each pending request from the internal status in the Service object (for that AsynchronousOperation) and executes the XML script associated to the Second operation for each request element having a pending status. If the execution is successful, the Polling Thread executes the XML script associated to the get request, using the result to build the SOAP response to be sent to the SOAP server (whose address was previously extracted from the SOAP Header of the SOAP request).
To send the response, an object of class AxisSOAPClient is used. Once sent the response, the related request element is removed from the Service internal status. Then the thread sleeps for a configurable amount of time
The TOOLBOX returns an error message in the following cases:

- A First script reporting the orderId of a pending request (for request incoming from SSE).
- A First script with an invalid XML payload (if the Schemas have been uploaded).
- upon any Second or Third script execution.
- Any internal error (IO errors and so on).
- The service is stopped

Validation is performed also on response data (if the Schemas have been uploaded).

If the asynchronous operation is supported by SSE in the first four cases an error message compliant with the SSE ICD is pushed to SSE. In case of operations not supported by SSE a SOAP fault is returned.
Operation initialisation can also includes the start of a thread (retryPush manager) performing re-tries on failed attempts to push responses to the SOAP Server (e.g. SSE). If the "retry" mechanism is enabled, the first time the push manager fails to push a response, it changes the corresponding request status in "responseLeaving". The retryPush thread performs an infinite loop in which it retrieves each request in "responseLeaving" status from the internal status in the Service object (the Operations belongs to) and tries to push the related response to the SOAP server. Every time a response is successfully pushed or the maximum number of attempts is reached the Service object removes the corresponding request its status.

One of the AsynchronousOperation configuration parameters is the request timeout. During the AsynchronousOperation initialisation a Thread checking expired timeouts is started. Its task is to check periodically the status of each request for the AsynchronousOperation (the status contains the time the request was received by the TOOLBOX). If a request is expired, it is removed from the list of the pending requests, no matter which status it is in.
The AsynchronouOperations objects, when the init() method is called, instantiate and start a Thread (Push manager) whose task is to check periodically the status and for each request in pending status and, possibly, to execute the check XML script. If the execution is successful it executes the Third XML script, packages the SOAP response, sends it to the SOAP server (using the AxisSOAPClient), e, g, SSE, whose address is extracted from the request SOAP header, and set the request to completed.
Additionally, an asynchronous operation can be configured so that if pushing response to the SOAP server (e. g. SSE) fails, a (configurable) number of new attempts can be performed at a configurable rate. Another thread (pushRetry manager) accomplishes this task. If the "retry" mechanism is enabled, the first time the push manager fails to push a response it changes the corresponding request status in "responseLeaving". The retryPush manager checks periodically the service status and for each request in "responseLeaving "status tries to push the related response (until the maximum number of attempts is reached or the response is successfully pushed). When whether even the last attempt fails or the push succeeds, the retryPush manager set the status of the request to unpushed.
The TOOLBOX extracts the service the request is addressed to from the URI of the request itself (which has the format http://<host>[:port]/TOOLBOX/services/<serviceName>). From the SOAPAction in the HTTP header it extracts the operation to execute. The TOOLBOX executes the (Jelly or TOOLBOX XML) script related to the requested operation and sends back the response in the same HTTP exchange.
The TOOLBOX returns an error message in the following cases:

- The service is stopped.
- The request contains an invalid XML payload (if the validation is turned on).
- A script execution error occurs
- Any internal error (IO errors and so on).

Validation is performed also on response data (if turned on).
In case of script error execution the TOOLBOX returns an error message compliant with the SSE ICD for all operations supported by SSE. In the other cases and for operation not supported by SSE a SOAP Fault is returned.

4.1.3.4. Resources

The status of each request (for asynchronous operations) is maintained in a file on disk. During the initialisation of the Service this file is loaded and the status of each request is restored. During Service operations the service maintains a RAM copy of this status (a DOM Document). Each time the document is modified it is dumped on disk so that persistence between different TOMCAT runs is achieved. Each request can be in one of the following status:

- **waiting**: the request is arrived but the First script has not yet been executed (for example because the queuing mechanism is on, or the service is suspended in SOFT mode)
- **executing**: the First script is executing (if a request is found to be in that status during initialisation, its state becomes aborted, because Tomcat could have been shut down while the script was running)
- **pending**: the First script has been successfully executed, but yet no successful Second script has been executed
- **ready**: a successful Second script has been executed (this status is important because it allows the TOOLBOX to guarantee the Service Provider that only ONE successful Second script is executed)
- **aborted**: a request is in this state if an error occurred during the First script or has been loaded as executing during initialisation
- **cancelled**: the request has been cancelled by the TOOLBOX administrator
- **rejected**: the request has been rejected because the Service is suspended in HARD mode.
- **expired**: the request permanence in the service status has exceeded the timeout (the timeout amount is a service configuration parameter).
- **responseLeaving**: a request is in this status when pushing the related response failed and a new attempts are foreseen.
- **completed**: a request is in this status when the response message has been sent to the SOAP server.
- **unpushed**: a request is in this status when pushing the related response failed and no additional attempts are foreseen.

4.1.4. ToolboxEngine

4.1.4.1. Type

The ToolboxEngine has been developed as a Java class,

4.1.4.2. Purpose

The ToolboxEngine is responsible for the XML scripts execution.

4.1.4.3. Function

It implements the TagInterpreter interface, which contains the getObject method which is the entry point for the interpretation of a tag. A Service object when needed invokes its executeScript method. For each available tag, the ToolboxEngine has an associated method, so that the execution of a script consists in a recursive navigation and execution of the DOM structure of the XML script. Each method returns a Java object. The object returned by the executeScript method is the one returned by the method associated with the root element of the script executed (see the SUM for a description of the operation performed and the object returned by each of the available tags).

4.1.5. Administration WEB Application

4.1.5.1. Type

The administration web application has been developed as a set of JSP pages

4.1.5.2. Purpose

The purpose is to provide an efficient tool to let the administrator manage the Toolbox RE and its services.

4.1.5.3. Function

Some JSP pages that access the running TOOLBOX servlet to perform administration tasks compose the WEB Application used to administrate the TOOLBOX. The TOOLBOX methods invoked by the JSP shall manage the list of the currently deployed services and the basic mechanisms of every service operation. We can foresee two kinds of operations: TOOLBOX operations and Service operation.
The first are administrative and monitoring common operations that act at a TOOLBOX level. The latter are administrative and monitoring operations that act at the Service level. Furthermore a secure access is provided via a login page.

4.1.5.4. Dependencies

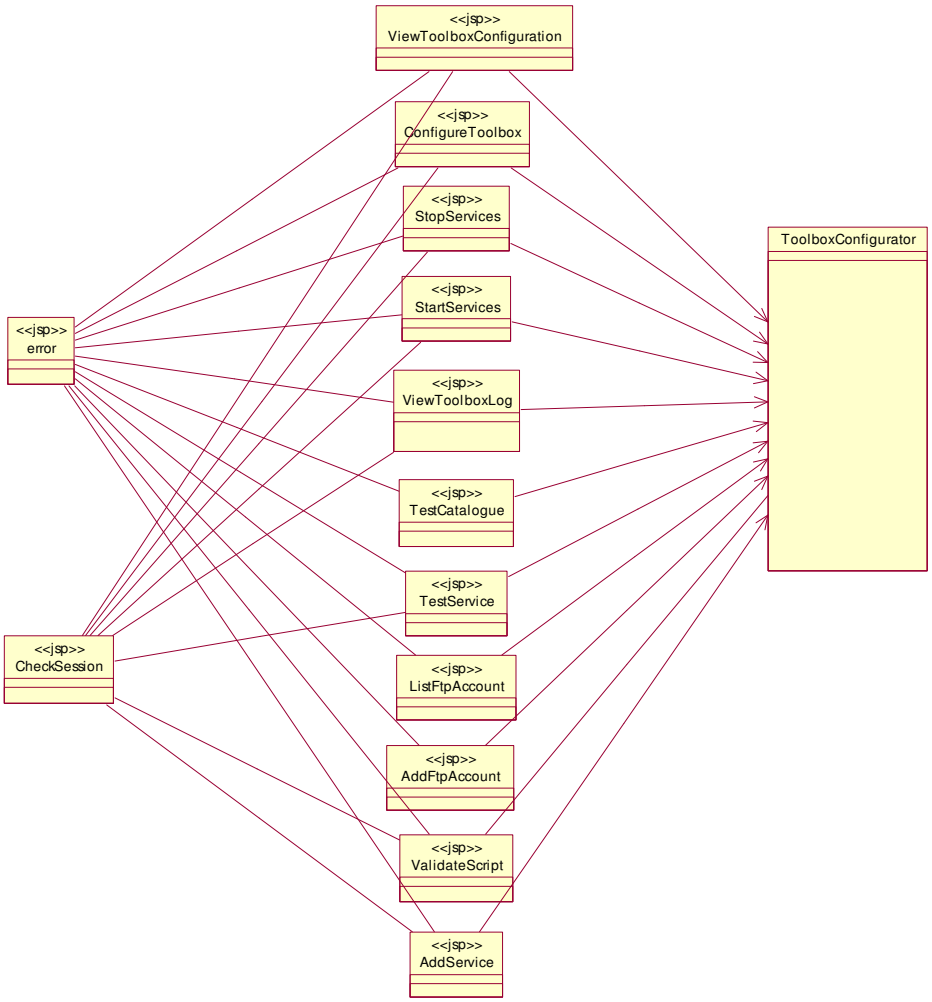

Figure 6 Toolbox Configurator and jsp relationships

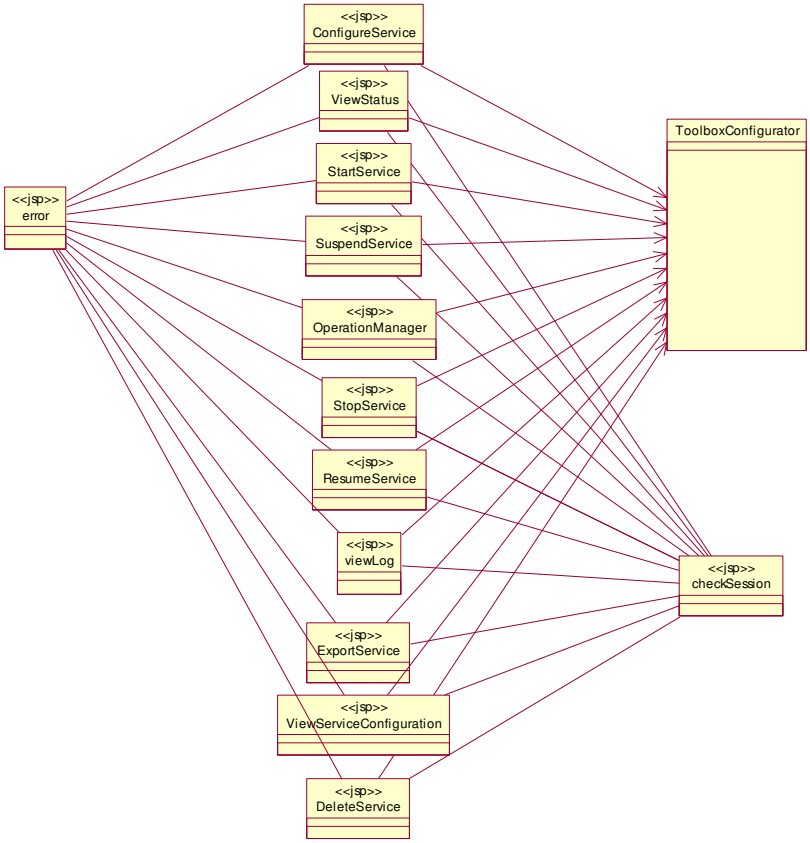

Figure 7 Check session and jsps relationships

4.1.6. Wizards

4.1.6.1. Type

Wizards are Java classes that have the structure described by Figure 8.

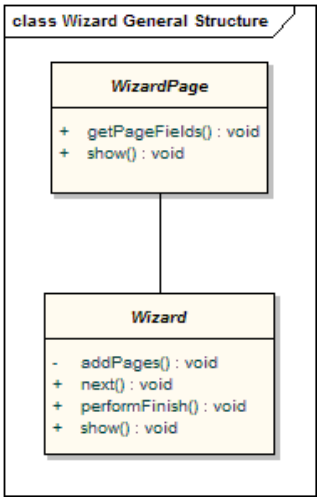

Figure 8 Wizard classes structure

4.1.6.2. Purpose

The purpose of a wizard is to collect several information from the user in order to complete a specific task.

4.1.6.3. Function

In this paragraph a generic description of a wizard is given, taking into account that the structure is the same for any wizard available into the Toolbox Development Environment.
Each wizard class is linked to Eclipse through the extension point org.eclipse.ui.newWizards, org.eclipse.ui.importWizard or org.eclipse.ui.exportWizard and performs the creation and configuration of all pages of the wizard through the method addPages().
The wizard is shown invoking the show() method and all its pages can be displayed invoking the next() method.
When the wizard job is completed, the method performFinish() is invoked and the task assigned to the wizard is performed, using the information stored in each wizard page.
During execution of the wizards, a check of the inserted values (like file name extensions etc.) is done in order to create valid and meaningful resource on the disk. This work is done by some of the helper methods implemented into the classes

4.1.7. LaunchDelegate

4.1.7.1. Type

LaunchDelegates are implemented as Java classes.

4.1.7.2. Purpose

The purpose of a LaunchDelegate is to execute the developed code in a specific mode (e.g. a Toolbox service on RUN or DEBUG mode).

4.1.7.3. Function

This class, see Figure 9, extends the org.eclipse.debug.core.launchConfigurationTypes extension point in order to specify the behaviour in case of testing a service operation.
The launch() method is used to execute an operation on the TOOLBOX Runtime Environment, behaving in different ways depending on the launch mode (currently only Run mode is taken into account). The method execute() of the class ToolboxScriptRunLaunch is invoked in order to perform all testing steps, described below:

1. Connecting to the TOOLBOX Runtime Environment
2. Checking the service existence
3. Updating the operation details
4. Testing the service sending an input message.
5. Retrieving and storing all response messages.

4.1.7.4. Interfaces

Figure 9 LaunchDelegate class structure diagram

4.1.8. Script Editor

4.1.8.1. Type

The editor is developed as a Java class.

4.1.8.2. Purpose

Purpose of this class is to provide an editor for all Toolbox scripts.

4.1.8.3. Function

The TscriptMultiPageEditorPart class implements the editor for all files of type tscript. It is an extension of the XMLMultiPageEditorPart, inheriting all characteristics and adding some helper methods for use into the code of other classes. All main features of this editor are implemented by the XMLMultiPageEditorPart class, in order to inherit already existing features and not to implement it. This process also benefits from all patches and features enhancements of the WTP plugin (that contains the XMLMultiPageEditorPart class).

4.1.8.4. Interfaces

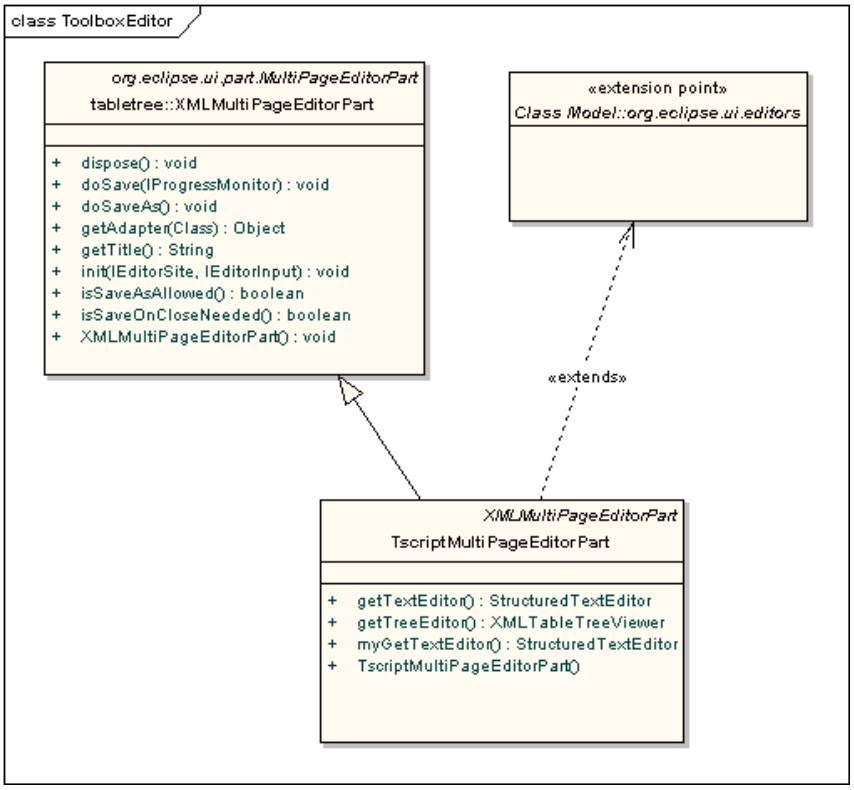

Figure 10 Toolbox Editor class diagram

4.1.9. Text Hover

4.1.9.1. Type

Text Hover is implemented as a Java class and is connected to Eclipse via a specific extension point.

4.1.9.2. Purpose

The purpose of TextHover is to provide information about the which over which the mouse pointer is hovering.

4.1.9.3. Function

The behaviour of the default editor for TOOLBOX scripts is modified through the org.eclipse.wst.sse.ui.editorConfiguration extension point and the TscriptMultiPageEditorPartConfiguration class. Through these, the editor can show documentation while hovering over a tag (in form of popup) and can format the XML script in a manner that doesn't introduce any error.
When the user hovers over a tag, the method getTextHover of the class TscriptMultiPageEditorPartConfiguration is invoked in order to obtain the area to allocate for the popup and to obtain its content. The area, retrieved by Eclipse through the method getHoverRegion, is calculated from the tag where hovering is done in order to show a popup just below it. The content is retrieved through an invocation of the method getHoverInfo, where it is extracted from the script schema through an XSL transformation.
When the user requests to format a TOOLBOX script, the method getContentFormatter is invoked and the TscriptMultiPageEditorTextFormatter class is obtained. The method format is invoked on this class and the text on a specified region (second parameter) is updated on the document (first parameter).

4.1.9.4. Interfaces

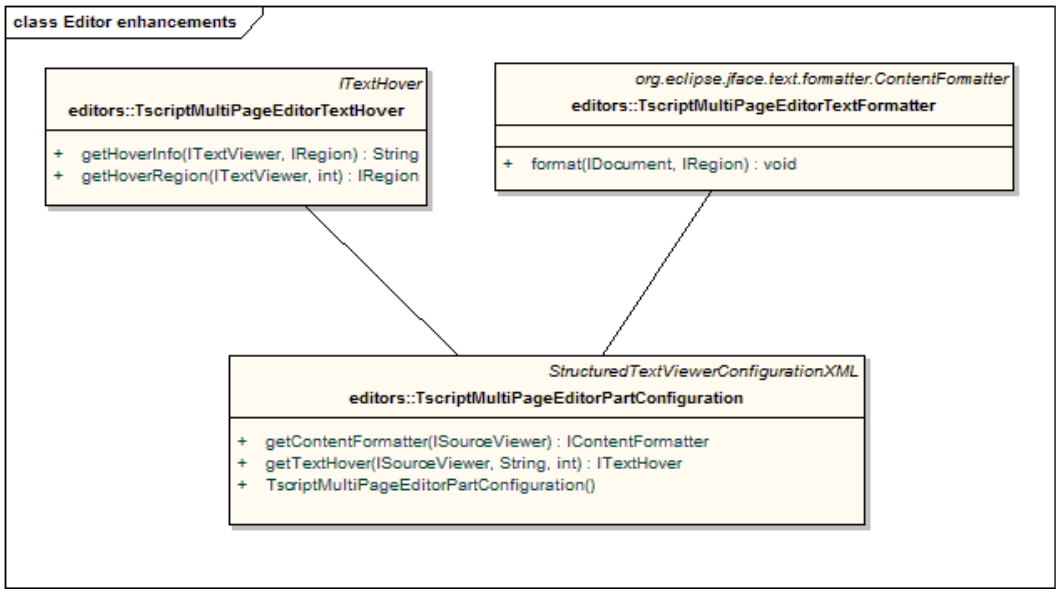

Figure 11 Text Hover class for Toolbox Editor

4.2. Internal Interfaces Design

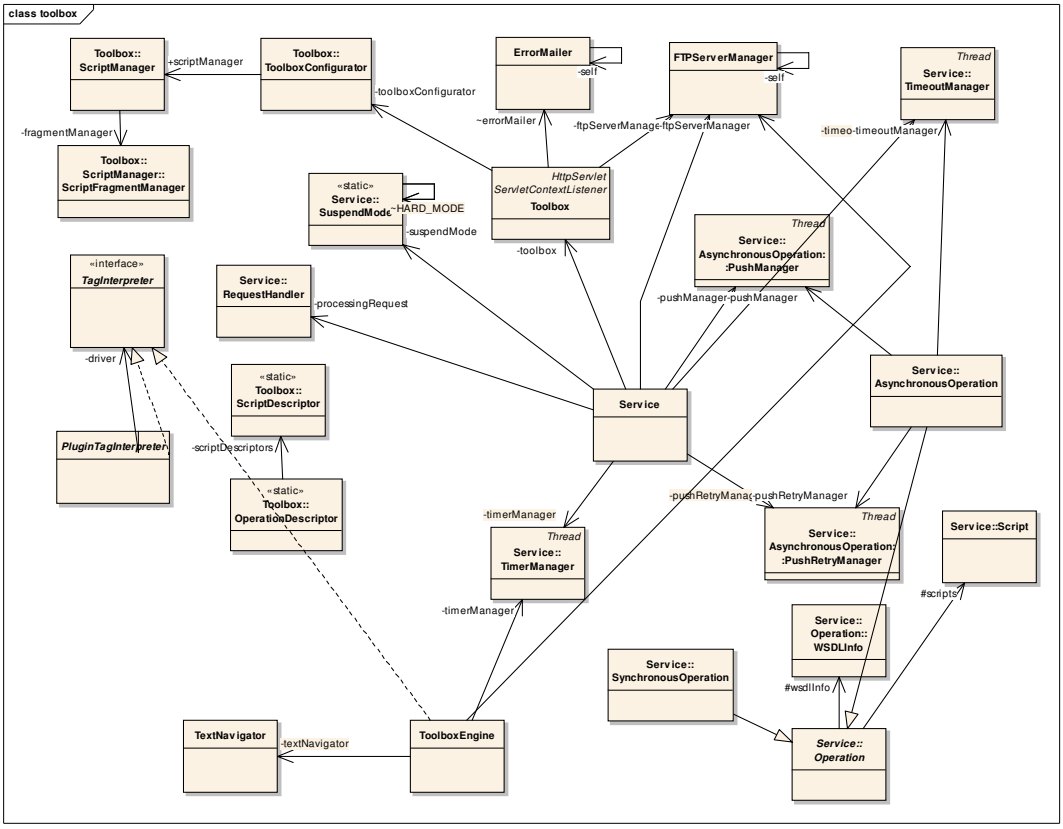

Figure 12 TOOLBOX RE internal dependencies and interfaces.

4.2.1. Internal Interfaces identification

4.2.1.1. Toolbox

The TOOLBOX servlet is deployed on TOMCAT and is accessible via HTTP through the URL: http://<tomcat_host>:8080/TOOLBOX/services/<service_name>.
Moreover the doPost method manages the remote service deployment. The remote service deployment functionality is accessible via the URL http://<tomcat_host>:8080/TOOLBOX/deploy/. Via this URL the TOOLBOX receives a SOAP message containing the export descriptor of the service to be published. Extract the descriptor from the message and deploy the service, if not already present.
In any moment there will be at most a single instance of this class running in TOMCAT, managing all the SOAP (over HTTP) requests.
Moreover the TOOLBOX exposes the interface http://<tomcat_host>:8080/TOOLBOX/manager which allows the client (and the Toolbox itself in some case) to operate with the internal data (services, log files etc). Via this entry point it is possible to send POST and GET requests in order to execute a task.

4.2.1.2. Requests execution

In the following we report a general sequence diagram both for synchronous and asynchronous operation in order to show internal relationship between components and their interfaces usages.

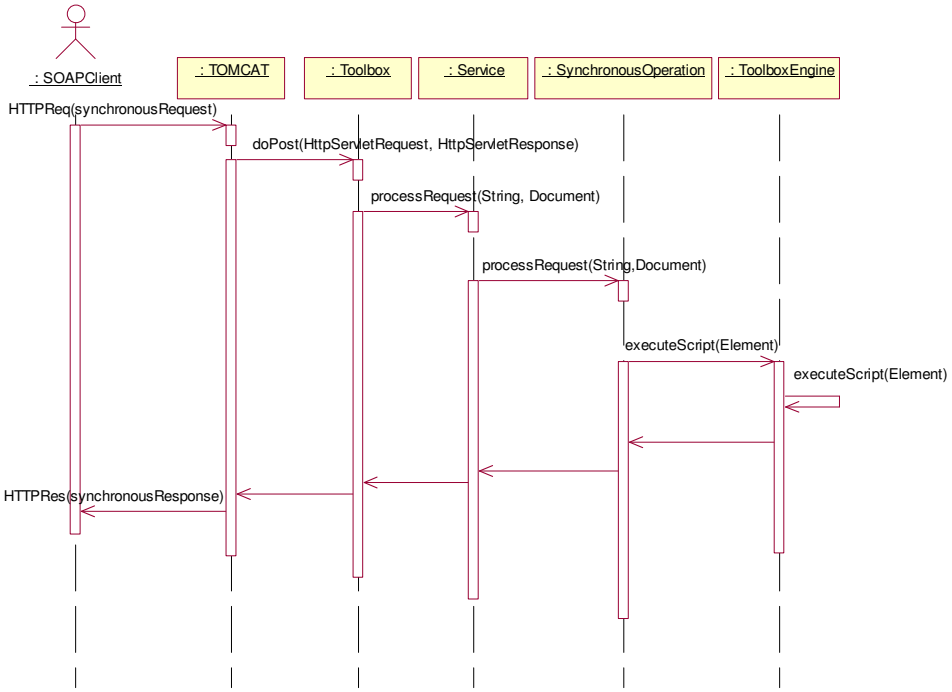

Figure 13 Synchronous operations

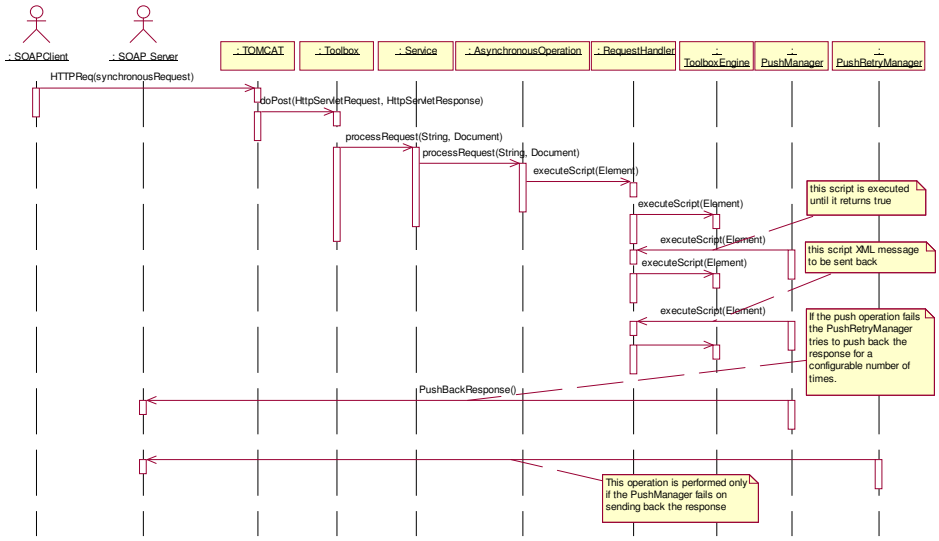

Figure 14 Asynchronous operations

4.2.1.3. Interaction with wizards

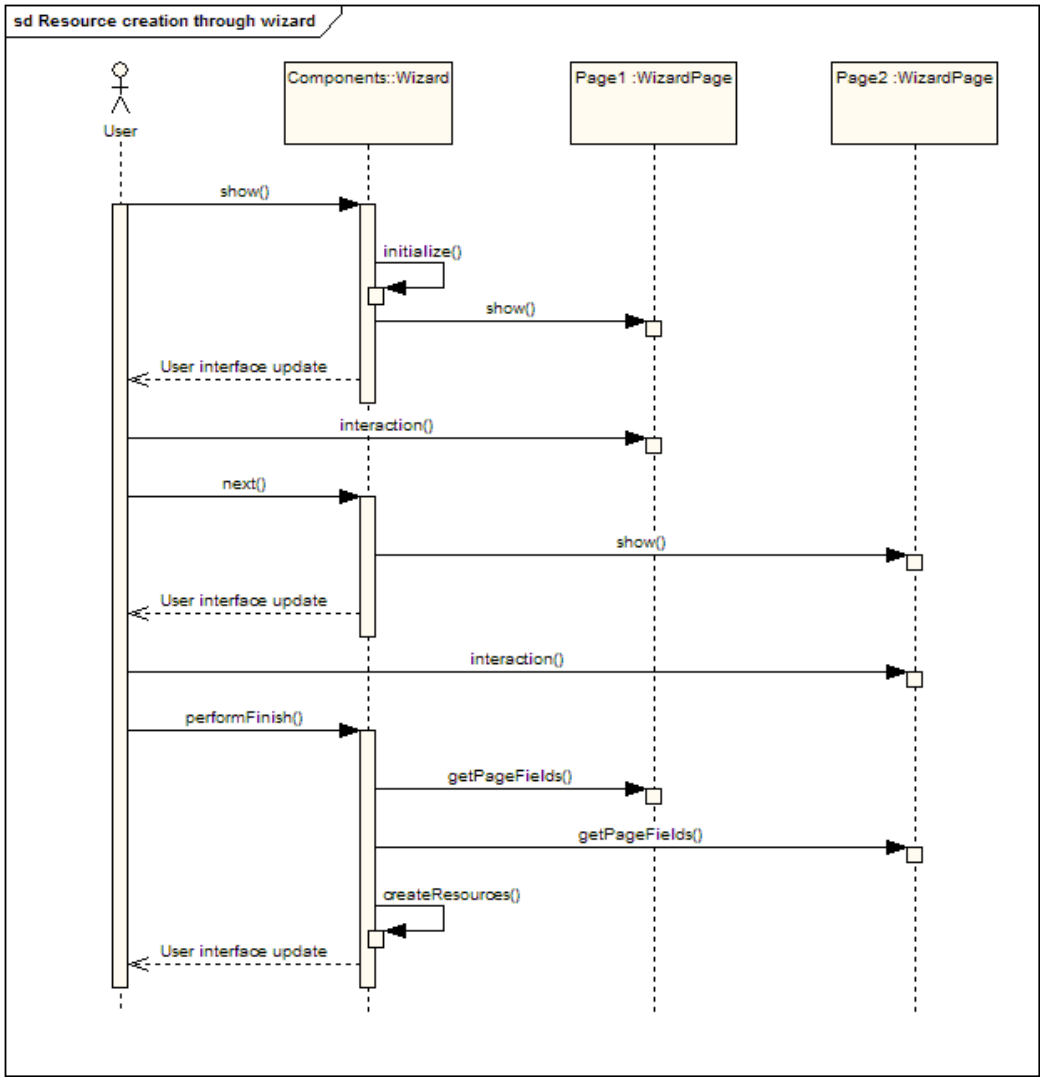

Figure 15 Sequence diagram showing interactions between user and wizards

Figure 15 shows how the user and wizard classes interact each other in order to provide the wizard paradigm. In this sequence diagram a general wizard with two pages is represented, but the diagram can be extended easily to any number of pages thanks to the generalization provided by the Eclipse code.
Looking at the sequence diagram, when the user selects a wizard the underlying code initializes itself and start showing the first wizard page. The user can then interact with all components provided by the page and, at the end, ask for the next page (if any). This action leads to the update of the GUI, displaying the following wizard page. The user can then interact with the new page and ask for the completion of the wizard. This action is

linked to a task performed by the wizard class in which all information are collected from each page and the resource is created.

4.2.1.4. Running a synchronous operation

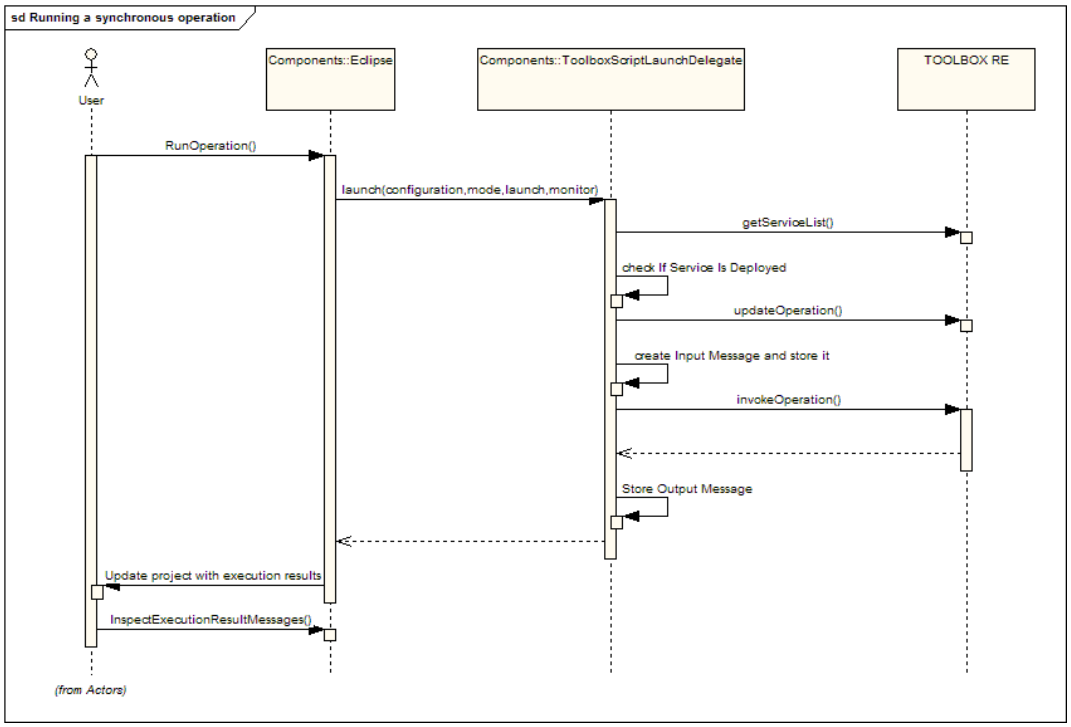

Figure 16 Sequence diagram of a synchronous operation execution

When starting the execution in run mode, the eclipse IDE instantiates the ToolboxScriptLaunchDelegate and calls its launch method. Inside this, a check against the TOOLBOX Runtime Environment is performed in order to verify that the service is deployed into it. If the user has modified the polling rate (setting it into the launch configuration), the operation on the TOOLBOX RE is updated. If all these steps are successful, the input message is created, stored into the project and sent to the TOOLBOX Runtime Environment. The answer is waited and finally store into the project.
When completed the workspace is updated and the user can inspect execution result files.

4.2.1.5. Running an asynchronous operation

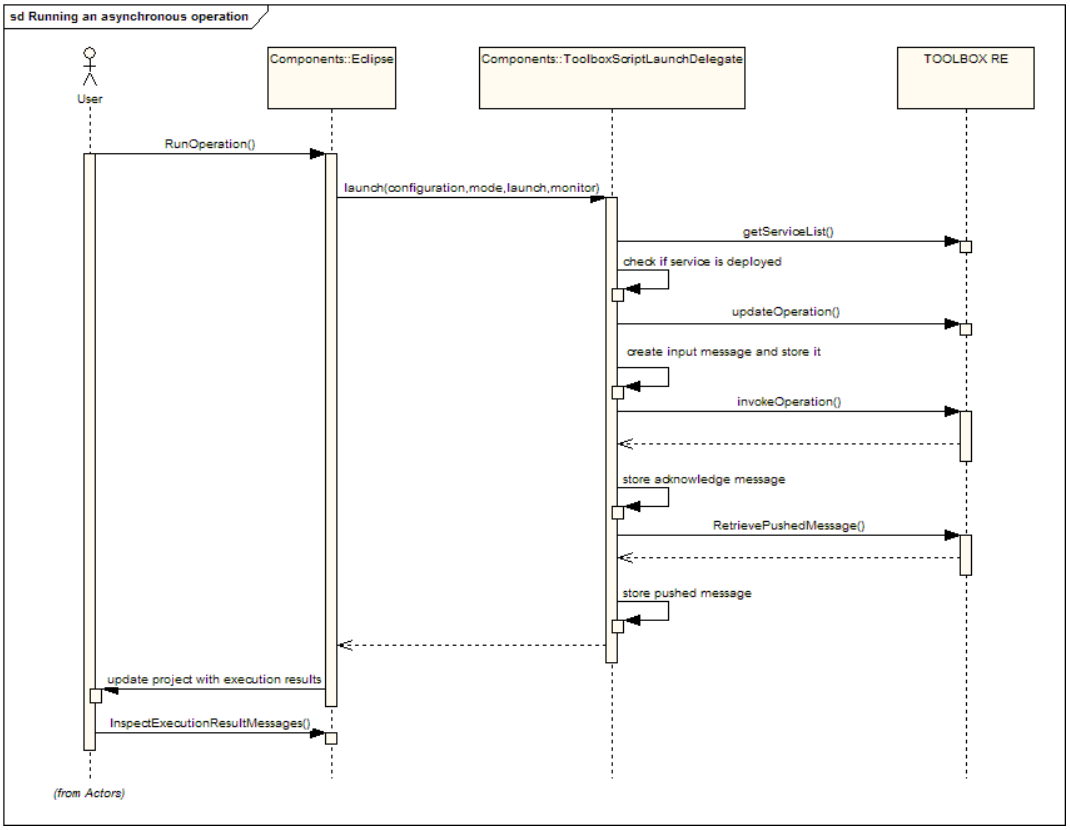

Figure 17 Sequence diagram of an asynchronous operation execution

Running an asynchronous operation is similar to executing a synchronous one, except for the fact that a loop is performed checking for the pushed message. This check is performed against the TOOLBOX Runtime Environment Push Server and, if the message is found, it is downloaded and stored locally for inspection.