CSOAP Architecture

Rafik CHIBOUT, Valérie ISSARNY, Daniele SACCHETTI.

Version 1.0
Feedback: rafik.chibout@inria.fr

Contents

1. Introduction **2**

2. CSOAP Server subsystems **2**

2.1. Transport subsystem *3*
2.2. Message model subsystem *4*
2.3. Encoding subsystem *9*
2.4. Provider subsystem *15*
2.5. Handlers subsystem *17*
2.6. Deployment subsystem *19*
2.7. Global Managing subsystem *24*
2.8. CSOAP Client subsystem *26*

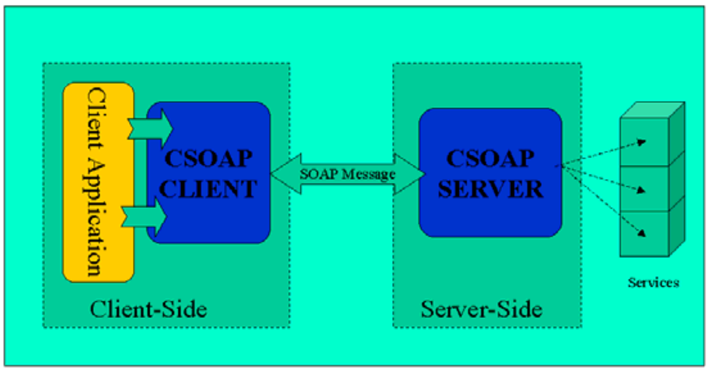
3. CSOAP Generator **28**

3.1. Introduction *28*
3.2. CSOAP Generator Architecture *29*
3.3. WSDL elements and java generated classes *30*

4. Appendix **35**

1. Introduction

CSOAP is essentially a SOAP engine for resource-constrained devices such as PDA (*Personal Digital Assistant*), which is able to deploy Web Services and to manage RPCs (Remote Procedure Call) from SOAP clients and dispatch them to services. CSOAP is also a framework for developing Web Services and their clients. CSOAP implementation follows the Sun's JAX-RPC Specification, which gives a standard for SOAP-based RPC to support the development of SOAP-based interoperable and portable Web services (see Appendix and http://java.sun.com/xml/jaxrpc/index.jsp for more details).



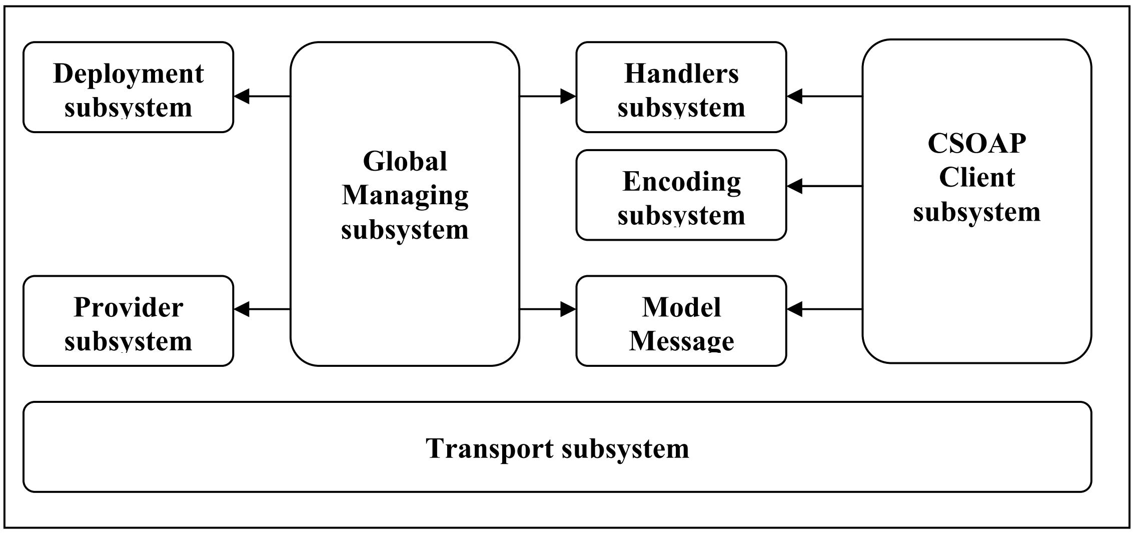
1. CSOAP Infrastructure

As shown by Figure 1 CSOAP is two parts: CSOAP Server part running into the Server-Side and CSOAP Client part running into the Client-Side. CSOAP Server manages the deployed services, listen to the SOAP messages come from Clients and dispatch them to services. CSOAP Client provides a set of methods which allows a simple development of Client Application that invokes some services deployed on CSOAP Server. CSOAP is organized into subsystems which make it modular and extensible. The follow Section will show these subsystems and will give their design.

2. CSOAP Server subsystems

In addition to the advantages of the Sun's JAX-RPC specification, CSOAP comprises several subsystems working together with the aim of separating responsibilities cleanly and making CSOAP modular. Subsystems enable parts of a system to be used without having to use the whole of it (or hack the code). Figure 2 shows the relationship between the different subsystems of CSOAP Server.

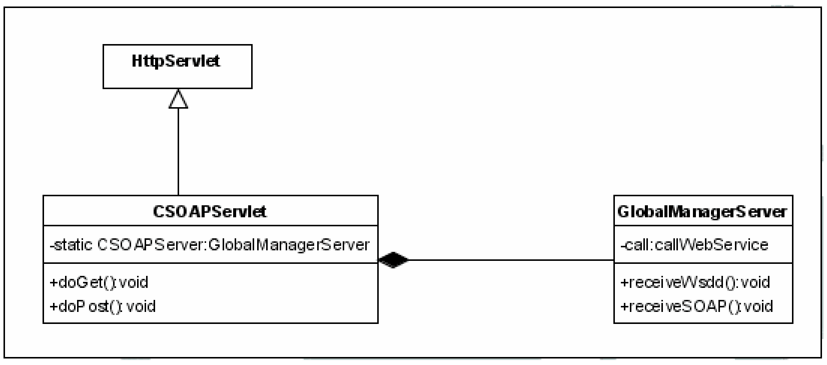
In this section we detail the CSOAP subsystems: the Transport subsystem that is charged of the reception and response over the transport protocol is described in section 2.1, the Model Message that implements the Sun JAXM specification for SOAP messages is described in section 2.2, the Encoding subsystem based on Sun JAX-RPC specification that translates XML data into java object and vice versa is described in section 2.3, the Handlers subsystem based on Sun JAX-RPC specification allows a further processing of the SOAP messages is described in section 2.5, the Provider subsystem that executes services is described in section 2.4, the Deployment subsystem that allows the deployment and configuration of services is described in section 2.6, the GlobalManaging subsystem that is charged of the coordination of the subsystems is described in section 2.7 and finally the CSOAP Client subsystem is based on the Model Message subsystem and provides the methods to make service calls to client applications is described in section 2.8.



2. CSOAP Server subsystems

2.1. Transport subsystem

The Transport subsystem is principally defined to guarantee the CSOAP independence towards the transport protocol. In CSOAP, only HTTP transport is actually supported. As you can see in Figure 3 the `CSOAPServlet` (provided by CSOAP) extends the class `HTTPServlet` and overrides the `doPost()` that receives the SOAP messages and the service deployment messages and forward them to the CSOAP Engine. A SOAP message is distinguished from a deployment message by the destination endpoint of the HTTP message: `AdminClient` in the case of the deployment and a service name the case of a SOAP message.

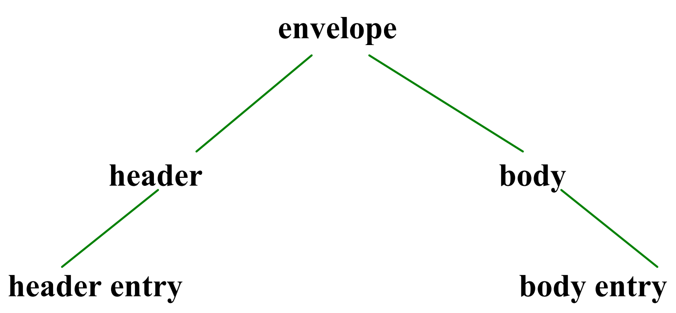


3. HTTP-Transport implementation

Figure 3 shows the relation between `CSOAPServlet` and the `GlobalManagerServer` class that is the entry point to the CSOAP Engine. The `receiveWsdd()` method of `GlobalManagerServer` is called when a service deployment file is received, and `receiveSOAP()` method of `GlobalManagerServer` is called when a SOAP Message is received.

2.2. Message model subsystem

This subsystem takes care of translating the data message received (or to be sent) into a standard model and it is based on the Sun's JAX-M Specification (`http://java.sun.com/xml/jaxm/index.jsp`).



4. Syntax of SOAP Message

As you can see in the figure above, the syntax of a SOAP message is fairly simple. A SOAP message consists of an *envelope* containing:

- an optional *header* containing zero or more *header entries*
- a *body* containing zero or more *body entries*

Some of the XML elements of a SOAP message define namespaces, each in terms of a URI and a local name, and encoding styles, a standard one of which is defined by SOAP.

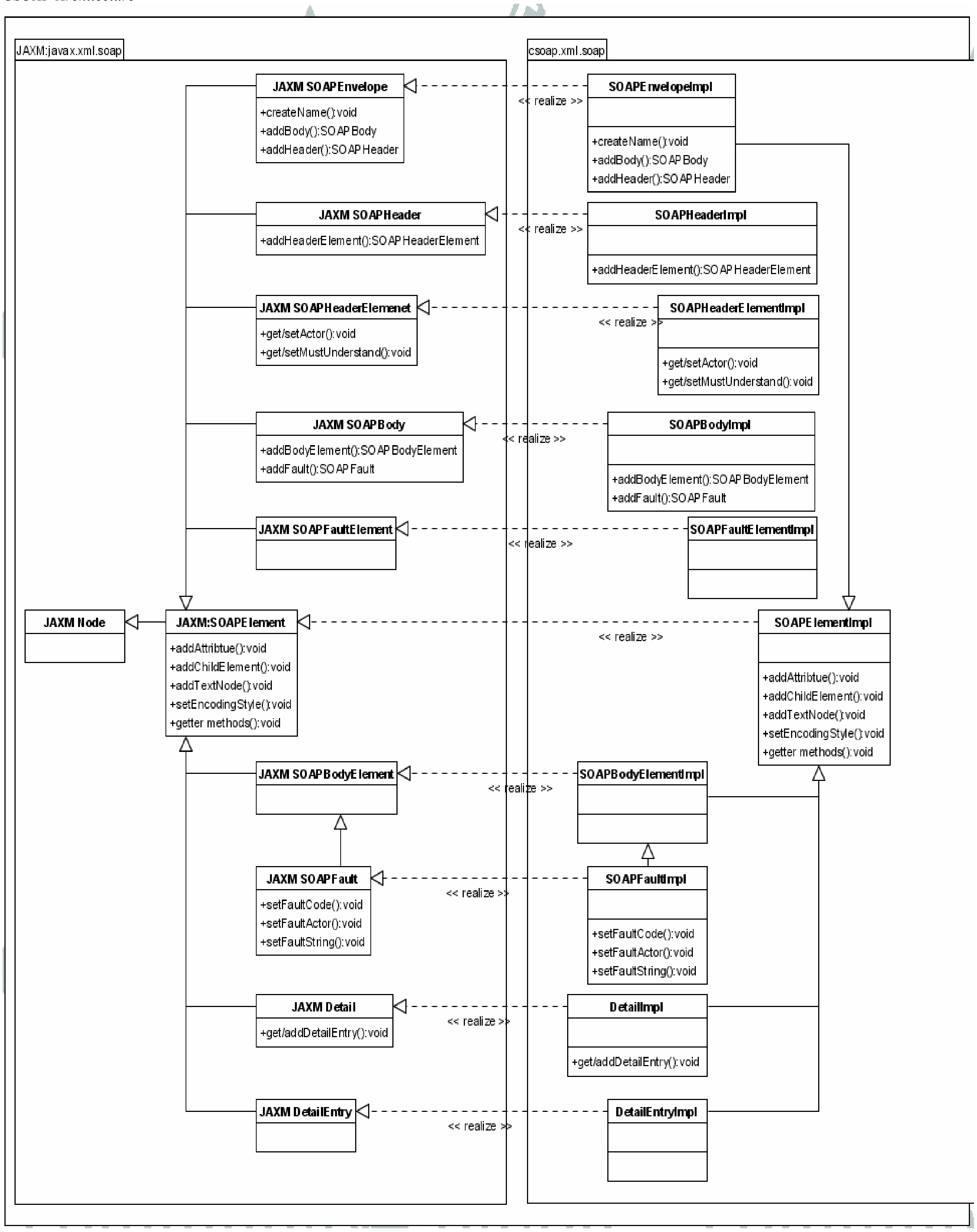
Header entries may be tagged with the following optional SOAP attributes:

- `actor` which specifies the intended recipient of the header entry in terms of a URI
- `mustUnderstand` which specifies whether or not the intended recipient of the header entry is required to process the header entry

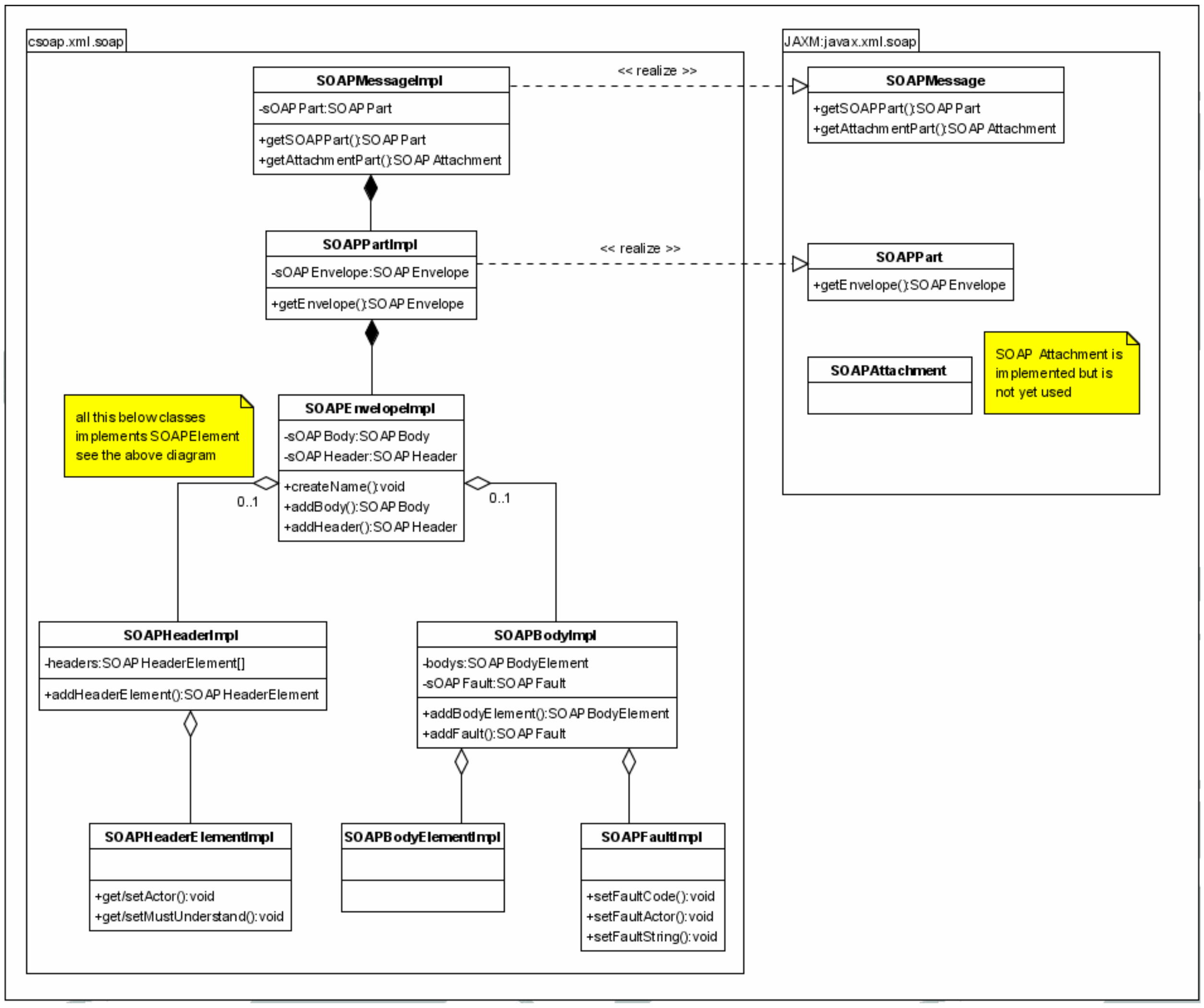
The body is the part of the SOAP message that contains the information required to invoke a service (i.e., the service name, method name and the parameter values). SOAP defines the fault entry that is a body entry and is used for reporting errors.

Each SOAP Message element implements the `javax.xml.soap.SOAPElement` interface which handles the element name, the attributes, the encoding style and the optional child elements. Look at the following UML schema in Figure 5.

Figure 6 shows how CSOAP implements the SOAP message structure seen in Figure 4 using the classes shown in Figure 5: a `SOAPMessage` object is created with a `SOAPPart` object that has a `SOAPEnvelope` object. By default, the `SOAPEnvelope` object has got an empty `SOAPBody` object and an empty `SOAPHeader` object. The `SOAPBody` object is required, and the `SOAPHeader` object, though optional, is used in the majority of cases. If the `SOAPHeader` object is not needed, it can be deleted. The access to the `SOAPHeader` and `SOAPBody` objects is given by the `SOAPEnvelope.getHeader()` and `SOAPEnvelope.getBody()` methods.

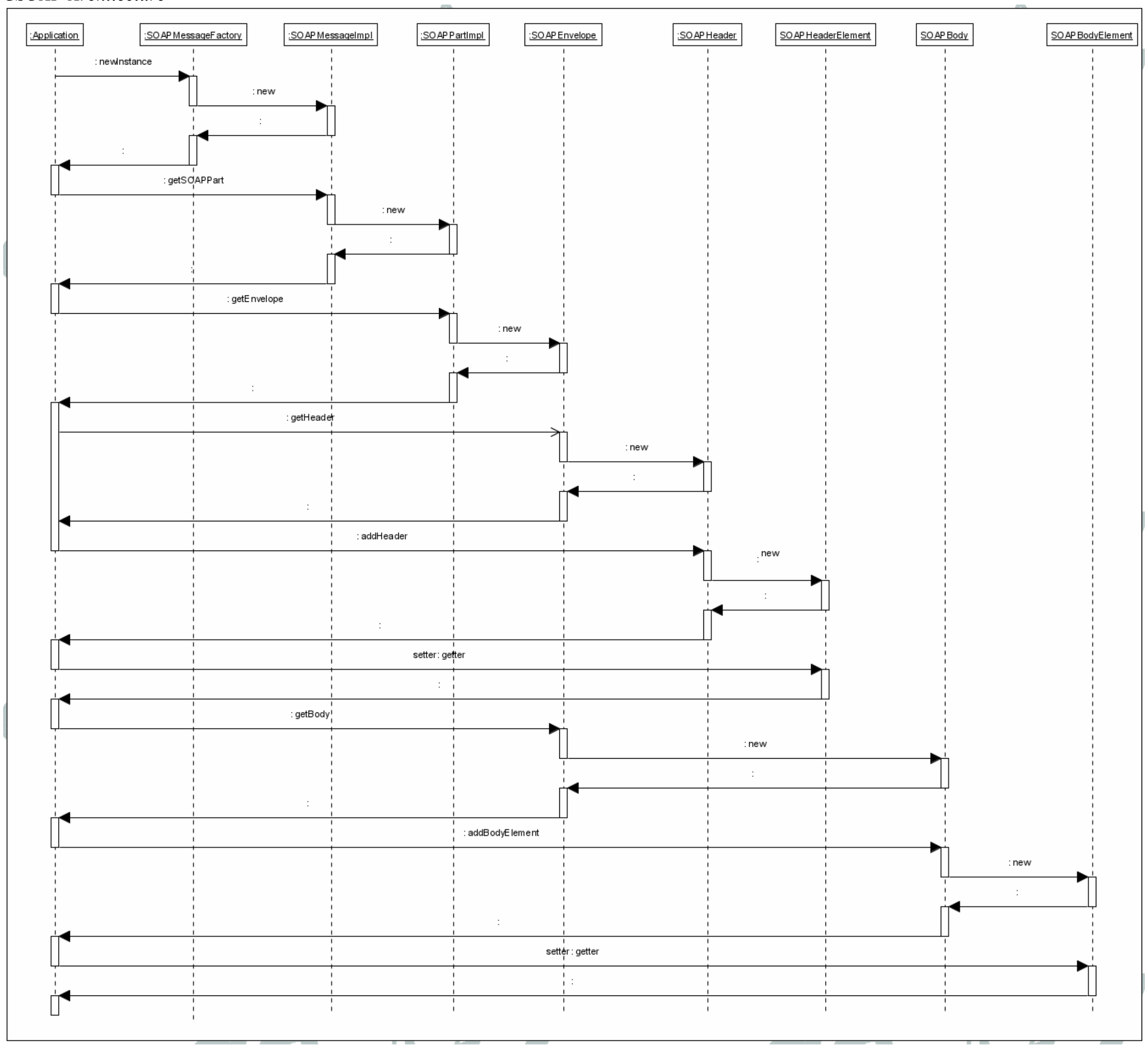


5. Class diagram of SOAP Message elements



6. Class diagram of SOAP Message

The sequence diagram in Figure 7 shows how a client can build a `SOAPMessage` object.
The `SOAPMessage` can be created by `newInstance()` method of `SOAPMessageFactory` class.
The `SOAPMessage` contains a `SOAPPart` which is returned by `getSOAPPart()` method.
The `SOAPPart` is used to retrieve `SOAPEnvelope`.
The `SOAPEnvelope` provides the `getSOAPBody()` method to retrieve a `SOAPBody` and `getSOAPHeader()` method to retrieve `SOAPHeader`.
The `SOAPBody` may create `SOAPBodyElement` that provides some getter and setter methods for the name, the type, and the text node and it further provides the possibility to create `SOAPElement` children.
The `SOAPHeader` may create a `SOAPHeaderElement` which provides some setter and getter methods for `actor` and `mustUnderStand` fields.



7. Sequence diagram of `SOAPMessage` Building

CSOAP uses an XML parser based on SAX and compliant with JAXP (*Java API for XML Processing, see* `http://java.sun.com/xml/jaxp/index.jsp`) to create a `SOAPMessage` starting from its XML representation.
Using JAXP compliant parsers involves implementing `org.xml.sax.helpers.DefaultHandler` interface. This interface allows catching events generated by the parser while reading XML files, when it detects the start or the end of an XML element. The generated event contains the name of the XML element along with its attributes.
In CSOAP the interface `org.xml.sax.helpers.DefaultHandler` is implemented by the class `csoap.xml.soap.SAXHandler` that receives in input the SOAP message as an `InputSource` object and builds a `SOAPMessage` object.
The class `csoap.xml.soap.SAXHandler` uses two state transition tables. In Figure 8 you can see the table that is examined when an XML start element is found and the `startElement()` method of the `csoap.xml.soap.SAXHandler` is invoked. The state will be updated taking into account the content of the table and the right object will be created and added to the `SOAPMessage` object. For example when the start of Envelope element is found, a `SOAPEnvelope` object is created and all its attributes and elements are added.

End element state table in Figure 9 is examined at the end of each XML element, when the `endElement()` method of the `csoap.xml.soap.SAXHandler` is invoked.

Stat/Element	Envelope	Header	Body	Fault	Any Element
0	1				
1		2	3		
2					4
3				7	6
4					4
5					
6			3		6
7					10
8					6
9					
10					10
11					

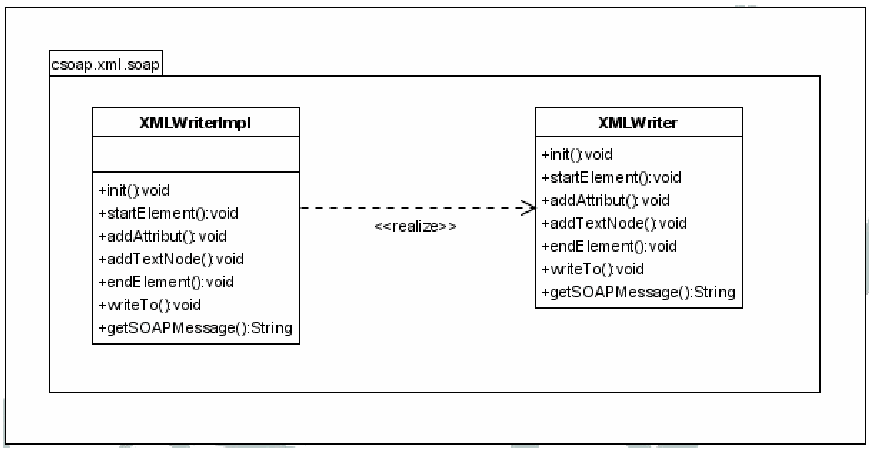
8. `State transition table (start element)`

Stat/Element	Envelope	Header	Body	Fault	Any Element
0	1				
1		2	3		
2					4
3				7	6
4					4
5					
6			3		6
7					10
8					6
9					
10					10
11					

9. `State transition table (end element)`

Sending a SOAP Message requires writing `SOAPMessage` on XML text representation, but the SAX parsers are not able to write an XML document. In CSOAP, this problem is resolved by `XMLWriter`, which allows writing a `SOAPMessage` Object into an XML text representation.
CSOAP provides the interface `csoap.xml.soap.XMLWriter` and its implementation `csoap.xml.soap.XMLWriterImpl` (see diagram in Figure 10) and the following is a brief description of their methods:

- `init()` initialize the class instance
- `startElement()` adds a new XML element and sets it as the current element. The parent of this new element is the current element
- `addAttribute()` adds an attribute to the current element
- `addTextNode()` adds a `textNode` text to the current element
- `endElement()` closes the current XML element and sets its parent as current element
- `writeTo()` writes the XML message in an `outputStream`
- `getSOAPMessage()` returns the XML message as a String



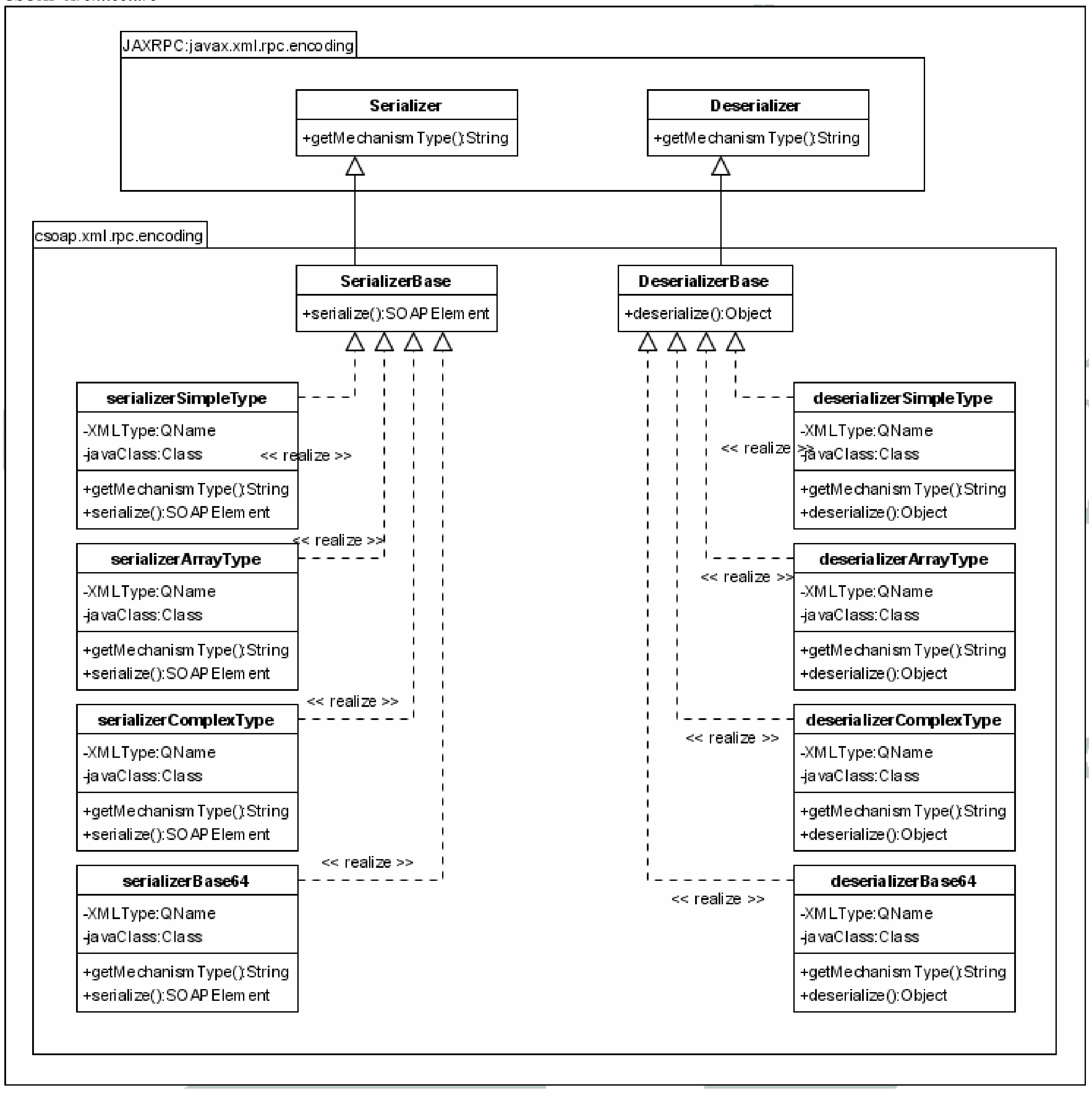
10. Class diagram of CSOAP `XMLWriter`

2.3. Encoding subsystem

This subsystem manages the mapping of Java data types into XML representations. In CSOAP, the encoding is called serialization and transforms Java objects and data of primitive Java types into XML data and decoding is called deserialization and transforms XML data into Java objects and primitive types. The UML schema in Figure 11 shows the serializer and deserializer classes provided by CSOAP Engine.

Each class implementing the interface `SerializerBase` or `DeserializerBase` contains the attributes `javaType` and `XMLType` storing respectively the Java Class and the XML type handled by the serializer/deserializer class.
The method `serialize()` provided by classes that implement the interface `SerializerBase` transforms a java object of type `javaType` into an object of class `SOAPElement` *(*which, is described in the above Section) of type `XMLType` by using a parsing mechanism.
The method `deserialize()` provided by classes that implement the interface `DeserializerBase` transforms a java object of type `SOAPElemen`t containing an XML element of type `XMLType` into an object of class `SOAPElemen`t of type `XMLType` by using a parsing mechanism.

As an example, the integer value 5 will be serialized by `serializerSimpleType.serialize()` into:
<x type xsd:int xmlns:xsd="http://www.w3.org/2001/XMLSchema"> 5 </x>
and contained in a `SOAPElementImpl` object with the following attribute values:
Name="x"
***type="{*http://www.w3.org/2001/XMLSchema:int}"**
textNode="5".



11. Classes diagram of serialisers and deserializers

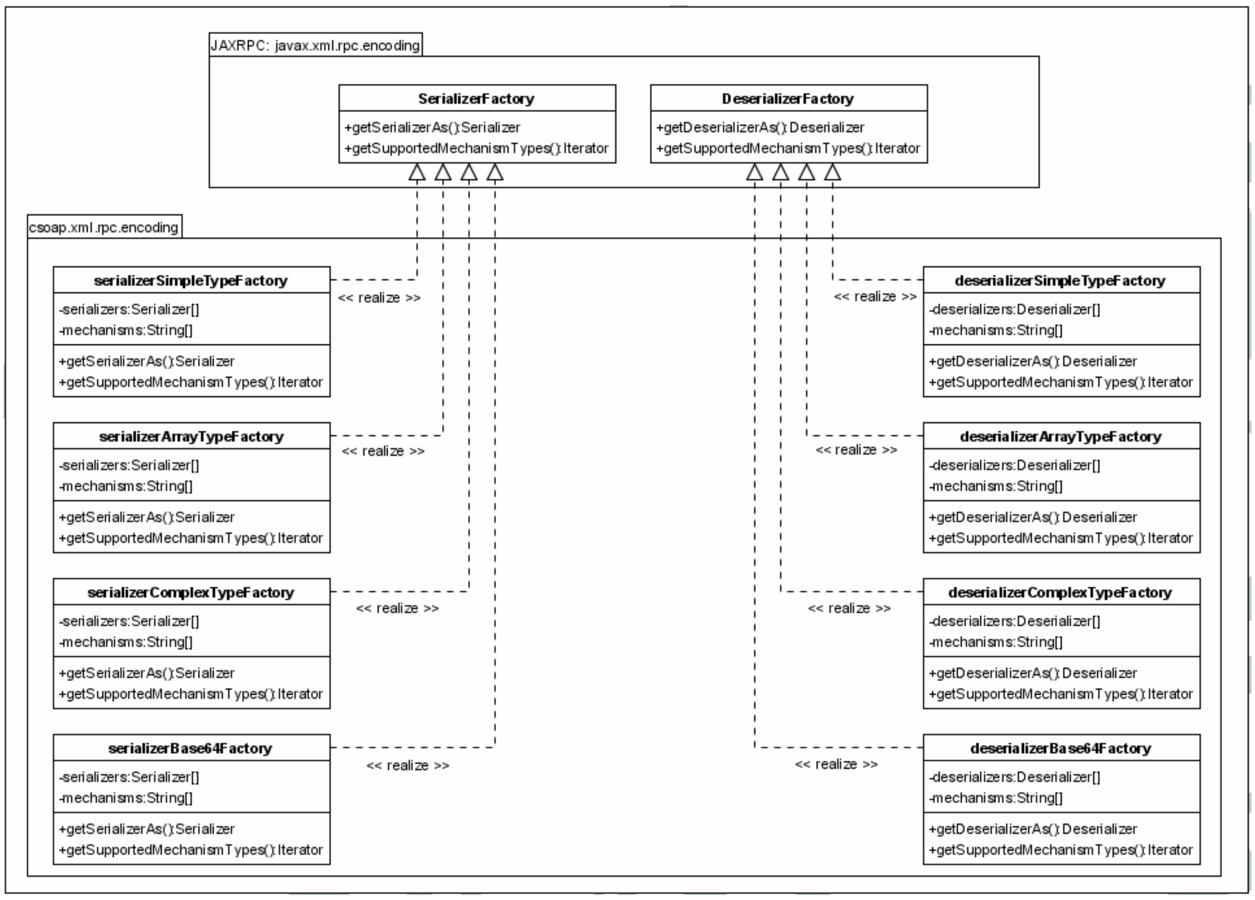
Here a more detailed description of the classes shown in figure 11:

- **`SerializerSimpleType`**: serializes all the primitive Java type (`int`, `long` …) into the correspondent XML.
- **`DeserializerSimpleType`**: deserializes all the XML simple types (`xsd:int`, `xsd:float`…) into their correspondent Java type.
- **`SerializerArrayType`**: serializes the one-dimension Java arrays containing elements of the same type, both simple and complex types (only those supported by `SerializerComplexType` and `DeserializerComplexType`).
- **`DeserializerArrayType`**: deserializes an XML type array `soapenc:array` into a Java array of elements of the type corresponding to the elements contained in the XML array.
- **`SerializerComplexType`**: serializes a Java class that contains only public attributes.

- **`DeserializerComplexType`**: deserialises an XML complex type into the corresponding Java class.
- **`SerializerBase64`**: serializes a binary data represented by an array of `byte` using base64 coding. This type is frequently used for the textual representation of binary files like images and videos.
- **`DeserializerBase64`**: deserializes an XML element coded in `base64` into a binary Java array.

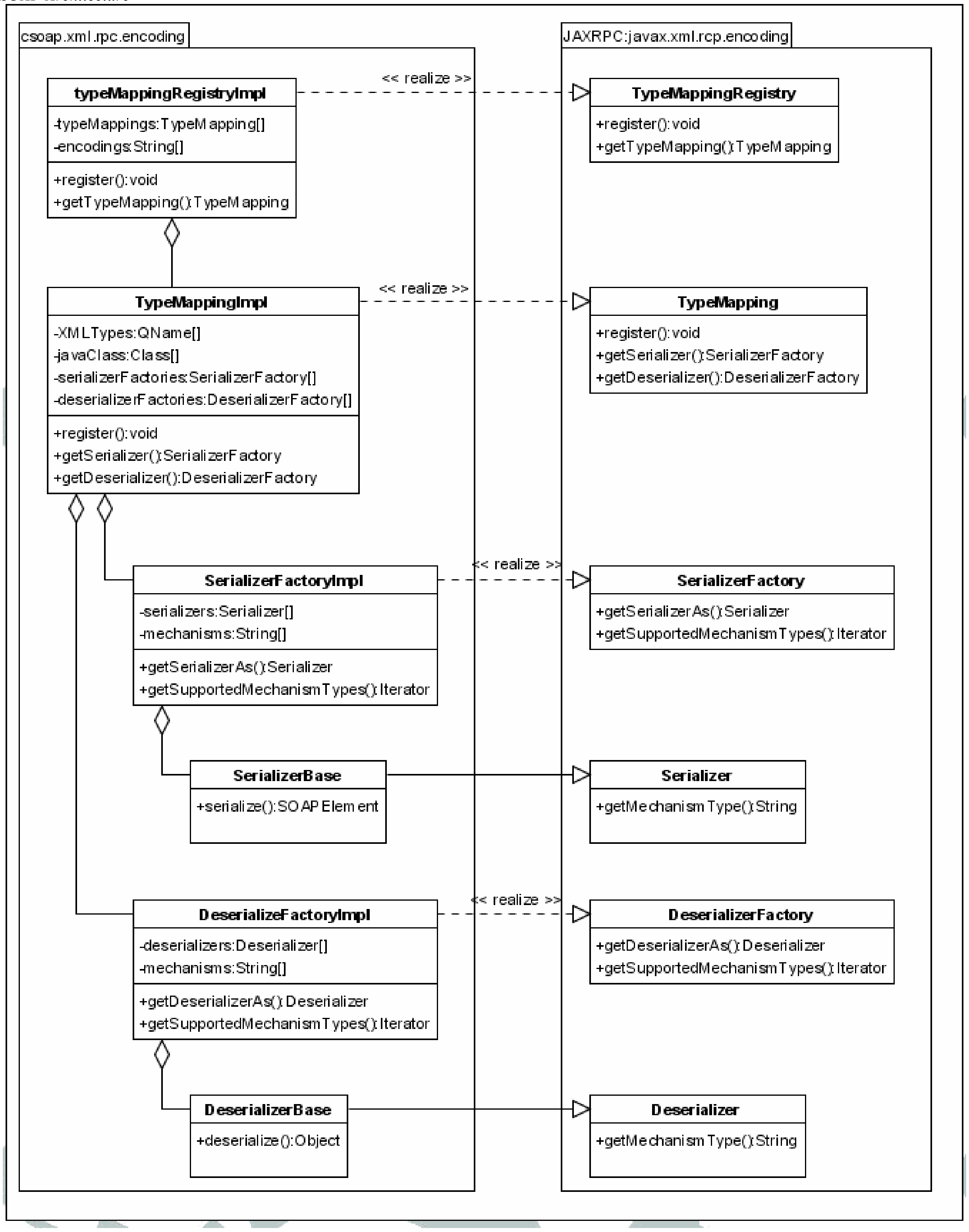
This serializers and deserializers provided by CSOAP uses only the SAX parsing mechanism, that is specified by "default" value of the mechanism attribute of serializer and deserializer classes.

All the serializer and deserializer classes are managed by a corresponding serializer or deserializer factory that has the role to store serializers and deserializers and to provide them to the SOAP components asking for them giving as input information the mechanism (in our case the "default" value of mechanism).
The following UML schema (Figure 12) shows serializer and deserializer factory classes provided by CSOAP Engine.


JAXRPC: javax.xml.rpc.encoding
SerializerFactory
+getSerializerAs():Serializer
+getSupportedMechanismTypes():Iterator
DeserializerFactory
+getDeserializerAs():Deserializer
+getSupportedMechanismTypes():Iterator
csoap.xml.rpc.encoding
serializerSimpleTypeFactory
serializerArrayTypeFactory
serializerComplexTypeFactory
serializerBase64Factory
-serializers:Serializer[]
-mechanisms:String[]
+getSerializerAs():Serializer
+getSupportedMechanismTypes():Iterator
deserializerSimpleTypeFactory
deserializerArrayTypeFactory
deserializerComplexTypeFactory
deserializerBase64Factory
-deserializers:Deserializer[]
-mechanisms:String[]
+getDeserializerAs():Deserializer
+getSupportedMechanismTypes():Iterator
<< realize >>

12. Classes diagram of serializer factories and deserializer factories

All the serializer and deserializer factories of the same XML encoding style are stored by a `typeMappingImpl` object that implements the interface `javax.xml.rpc.encoding.TypeMapping`.
All type Mappings are registered into a type Mapping Registry with their encoding style.
The `typeMappingRegistryImpl` class implements `javax.xml.rpc.encoding. typeMappingRegistry`. The type mapping class hierarchy is shown below (Figure 13).


csoap.xml.rpc.encoding
JAXRPC:javax.xml.rcp.encoding
typeMappingRegistryImpl
-typeMappings:TypeMapping[]
-encodings:String[]
+register():void
+getTypeMapping():TypeMapping
<< realize >>
TypeMappingRegistry
+register():void
+getTypeMapping():TypeMapping
TypeMappingImpl
-XMLTypes:QName[]
-javaClass:Class[]
-serializerFactories:SerializerFactory[]
-deserializerFactories:DeserializerFactory[]
+register():void
+getSerializer():SerializerFactory
+getDeserializer():DeserializerFactory
TypeMapping
+register():void
+getSerializer():SerializerFactory
+getDeserializer():DeserializerFactory
SerializerFactoryImpl
-serializers:Serializer[]
-mechanisms:String[]
+getSerializerAs():Serializer
+getSupportedMechanismTypes():Iterator
SerializerFactory
+getSerializerAs():Serializer
+getSupportedMechanismTypes():Iterator
SerializerBase
+serialize():SOAPElement
Serializer
+getMechanismType():String
DeserializeFactoryImpl
-deserializers:Deserializer[]
-mechanisms:String[]
+getDeserializerAs():Deserializer
+getSupportedMechanismTypes():Iterator
DeserializerFactory
+getDeserializerAs():Deserializer
+getSupportedMechanismTypes():Iterator
DeserializerBase
+deserialize():Object
Deserializer
+getMechanismType():String

13. Class diagram of `typeMapping`

The instances of classes implementing `SerializerBase` interface are stored in `SerializerFactoryImpl` according to the Java and XML type they handle. These instances are accessed through `getSerializerAs()` method of `SerializerFactoryImpl` by specifying a serializer mechanism.

Method `getSupportedMechanismTypes()` provides the list of mechanisms supported by the specific `SerializerFactoryImpl`.
The same class organization is used also for classes implementing the deserialization interface `DeserializerBase` and managed by `DeserializerFactoryImpl` classes.

`TypeMappingImpl` contains four arrays:

- `XMLTypes` containing XML types represented by `QName` type (`QName` represents a qualified name as defined in the XML specification. The value of a `QName` contains a Namespace URI, local part and prefix).
- `javaClasses` containing Class elements and representing Java types.
- `serializerFactories` containing `serializerFactory` elements.
- `deserializerFactories` containing `deserializerFactory` elements.

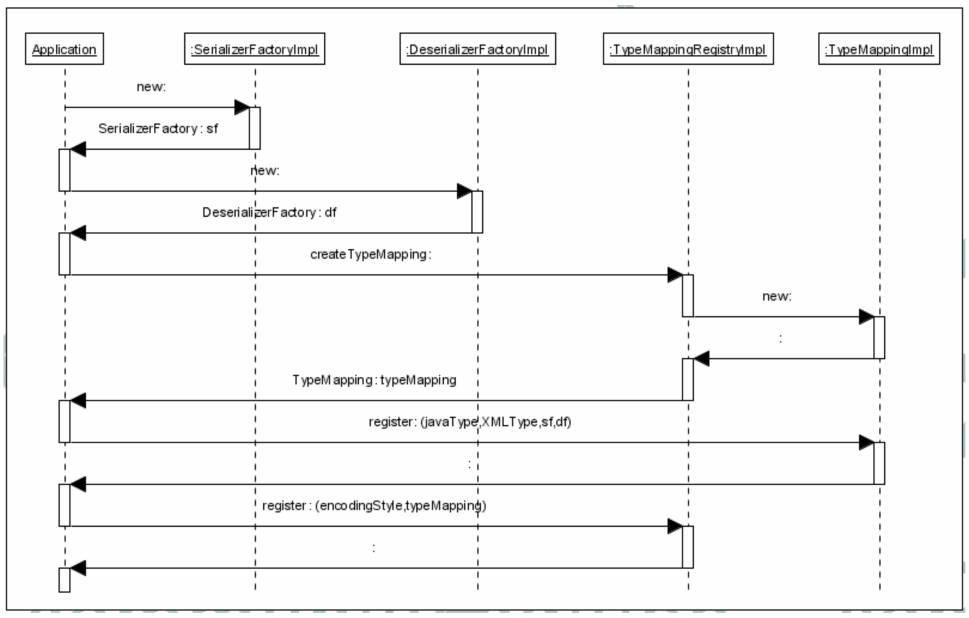
The XML type at position "i" in array `XMLTypes` corresponds to the Java type at postion "i" in javaClasses array. The `serializerFactory` and `deserializerFactory` store at the position "i", the serializer and deserializer corresponding to this couple of Java and XML types

`Register` method allows adding a new mapping by giving an XML type, a Java type and an instance of `SerializerFactory` and `DeserializerFactory` storing serializers and deserializers of Java and XML types.
Methods `getSerializer()` and `getDeserializer()` returns the serializer and deserializer factory instances of these Java and XML types given as parameters.

`typeMappingRegistryImpl` contains two arrays:

- `typeMappings` containing elements of `typeMapping` type.
- `encondings` containing String elements.

Each `typeMapping` stored at position "i" in array `typeMappings` corresponds to the coding style stored in encoding array at postion "i".
Method `register()` allows adding a new `typeMapping` and its corresponding encoding style.
Method `getTypeMapping()` returns the `typeMapping` corresponding to the encoding style given as parameter.

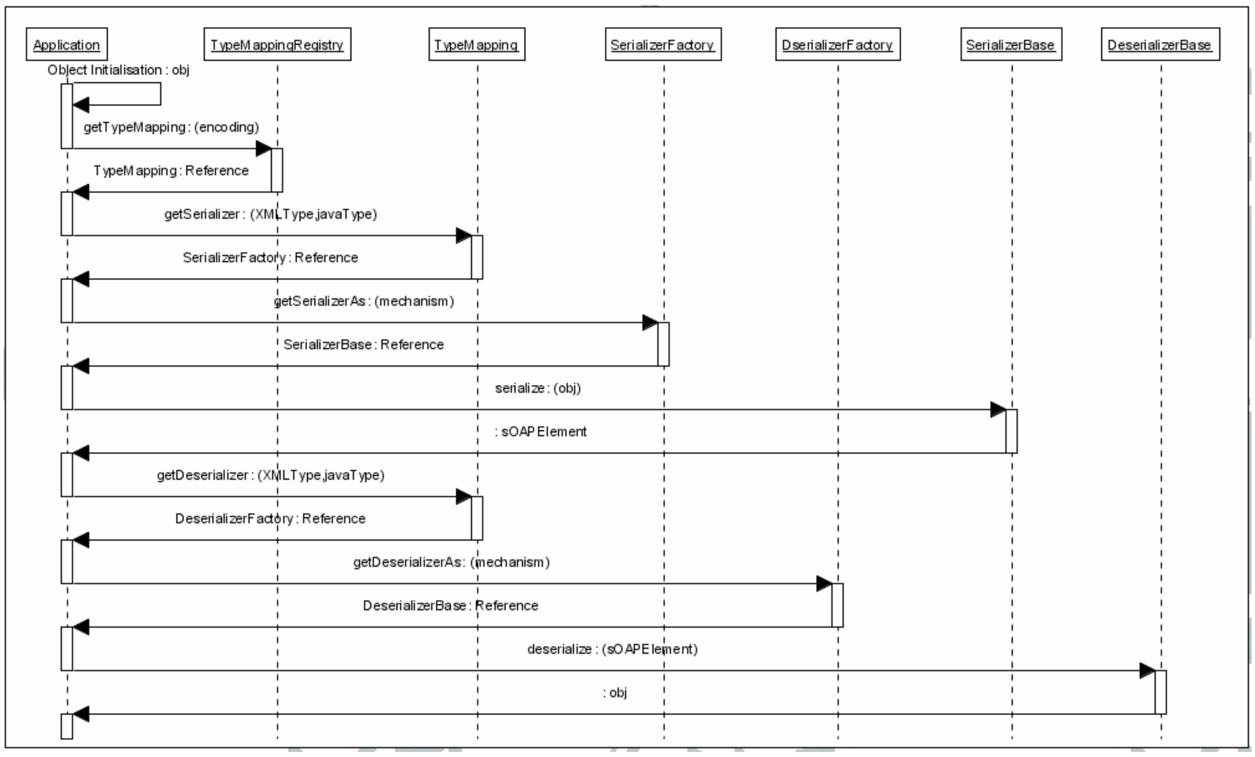


14. Create a new typeMapping

Sequence diagram in Figure 14 shows the creation and registration of a type mapping for a couple of Java and XML types. `SerializerFactoryImpl` and `deserializerFactoryImpl` (referenced with `sf` and `df`) store the serializer and deserializer aimed at mapping the Java and XML types and vice versa.

The application creates an instance of the class `serializerFactoryImpl` and of the class `deserializerFactoryImpl`. Then, it asks the `typeMappingRegistry` to create a new `typeMapping` with the call `createTypeMapping()`. This `typeMapping` is initialized with the right values of Java and XML types and the factory instances with the call `register()`.
Finally, the application registers the `typeMapping` and the encoding style in the `typeMappingRegistry` with method `register()`.

Sequence diagram in Figure 15 shows how an application can use the `typeMappingRegistry` to find the serializer and deserializer associated to a Java and an XML Types.


Application
TypeMappingRegistry
TypeMapping
SerializerFactory
DserializerFactory
SerializerBase
DeserializerBase
Object Initialisation : obj
getTypeMapping : (encoding)
TypeMapping : Reference
getSerializer : (XMLType,javaType)
SerializerFactory : Reference
getSerializerAs : (mechanism)
SerializerBase : Reference
serialize : (obj)
: sOAPElement
getDeserializer : (XMLType,javaType)
DeserializerFactory : Reference
getDeserializerAs: (mechanism)
DeserializerBase : Reference
deserialize : (sOAPElement)
: obj

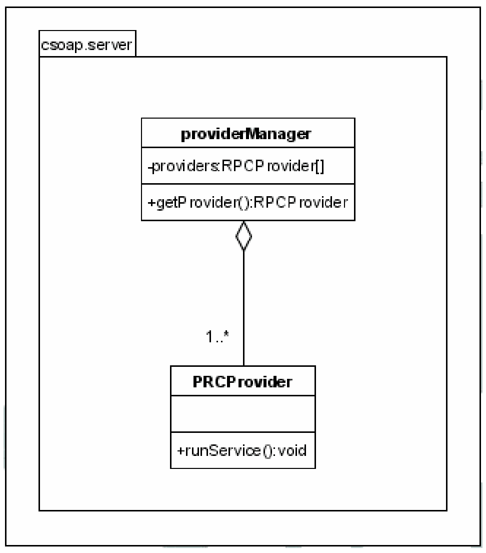
15. Using `TypeMappingRegistry`

The application makes use of the method `getTypeMapping()` of the `typeMappingRegistry` to retrieve the encoding type it wants to use. The `getSerializer()` method of `typeMapping` provides the `serializerFactory` for the Java (and/or XML) type specified.
The method `getSerializerAs()` of `serializerFactory` allows to retrieve the serializer by giving the parsing mechanism value associated to it (in our case "default"). Then, the application can call the `serialize()` method to execute the serialization: the input parameter is the Java object and it returns a `SOAPElement` object containing the XML representation of the input object.
The same process is used for deserialization. `Deserialize()` method has a `SOAPElement` object in input and returns a Java object containing the value stored by the `SOAPElement` object.

All the applications can create an instance of class TypeMappingRegistryImpl, register a new mapping between a Java type and an XML type and use this mapping or the already registered mappings to serialize Java objects and deserialize XML elements.

2.4. Provider subsystem

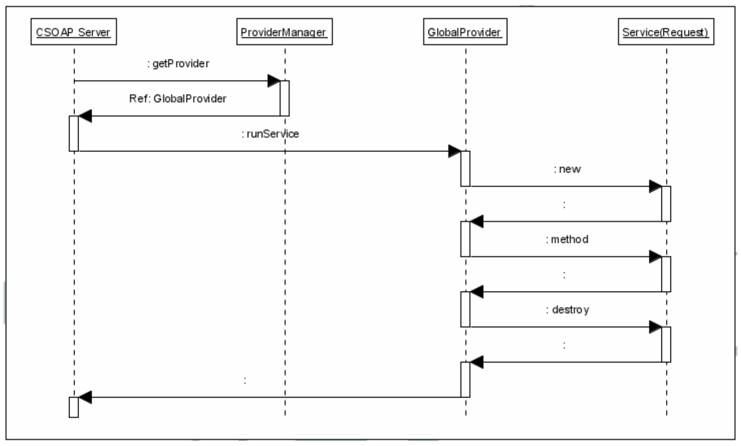
The provider is responsible for loading a Service instance and performing the requested Service method. The Provider subsystem takes care of managing the scope type with which the provider runs the service. The possible deployment scopes for a service are *Session* scope, *Application* scope or *Request* scope.
The services deployed with Application or Request scopes will be executed by a GlobalProvider which is a global instance of RPCProvider class.
An instance of RPCProvider class is created for each client that invokes a *Session* service. The session is maintained between the client and CSOAP Engine by an identifier given by CSOAP Engine and exchanged into the HTTP cookies.
The getProvider() method selects (or creates) and return an adequate provider able to run the requested service. If the service is deployed with a specified user-defined provider, the provider subsystem creates an instance of this user-defined provider and manages it like the RPCProvider class instances. The following (Figure 16) shows the class diagram of this subsystem.



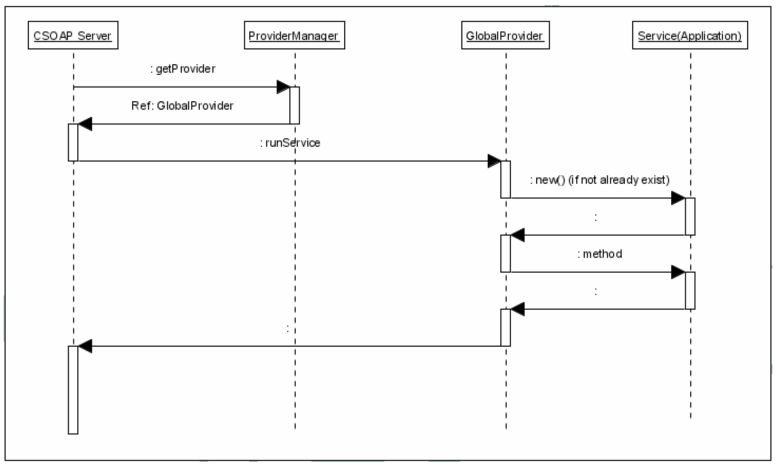
16. Class diagram of Provider subsystem

The instance of providerManager contains an instance of class RPCProvider called GlobalProvider that has is charged of the invocation of services deployed with scope Application and Request. New instances of RPCProvider class are created when services are deployed with Session scope. The three following diagrams show the different way ProviderManager handles a service according to its deployment scope.

Figure 17 shows how ProviderManager handles a client request when the called service is deployed with request scope. ProviderManager.getProvider() returns a reference to GlobalProvider. CSOAP Server invokes the method runService() of GlobalProvider giving all the information required for the service invocation. GlobalProvider creates an instance of service, invokes the method and finally destroys the service instance. To sum up, one instance of the service is created for each service call.



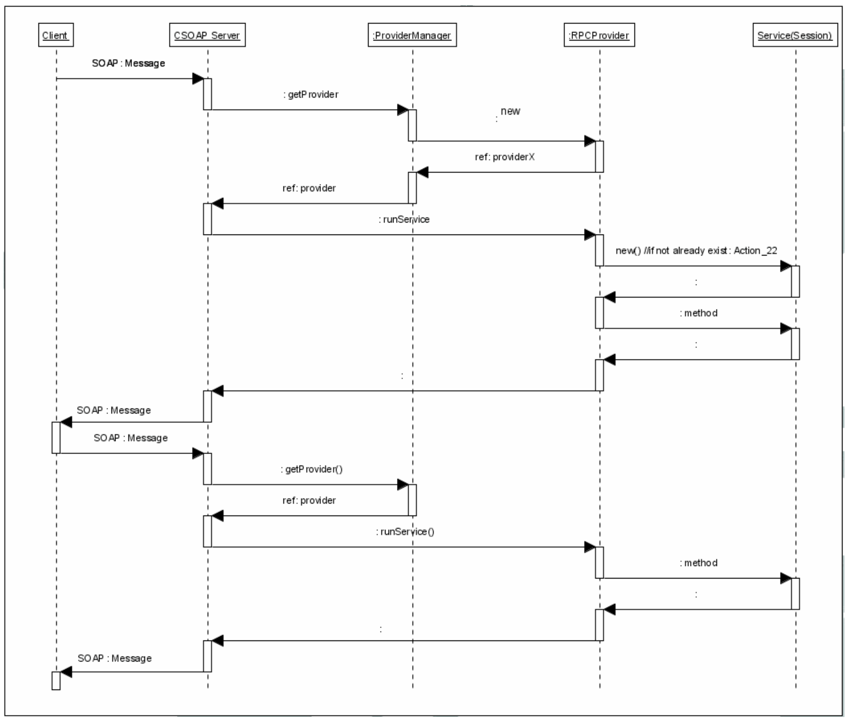
17. Sequence diagram of ProvideManager functioning (Service Request)



18. Sequence diagram of `ProvideManager` functioning(Service Application)

Figure 18 shows how `ProviderManager` handles a client request when the called service is deployed with Application scope. `ProviderManager.getProvider()` returns a reference to `GlobalProvider`. CSOAP Server invoke the method `runService()` of `GlobalProvider` that creates an instance of the service if an instance of the service is not already available.

Figure 19 shows how `ProviderManager` handles a client request when the called service is deployed with session scope. `ProviderManager` creates an instance of `RPCProvider` that is bound to the client.
CSOAP Server invoke the method `runService()` of `RPCProvider` giving all the information required for the service invocation.
`RPCProvider` creates an instance of service if an instance of the service is not already available.
All the service calls coming from the same client for this service are executed on the same instance of service, unless the session between the client and CSOAP server has expired. The duration of the session is fixed by the `delay` variable of class `ProviderManager`.



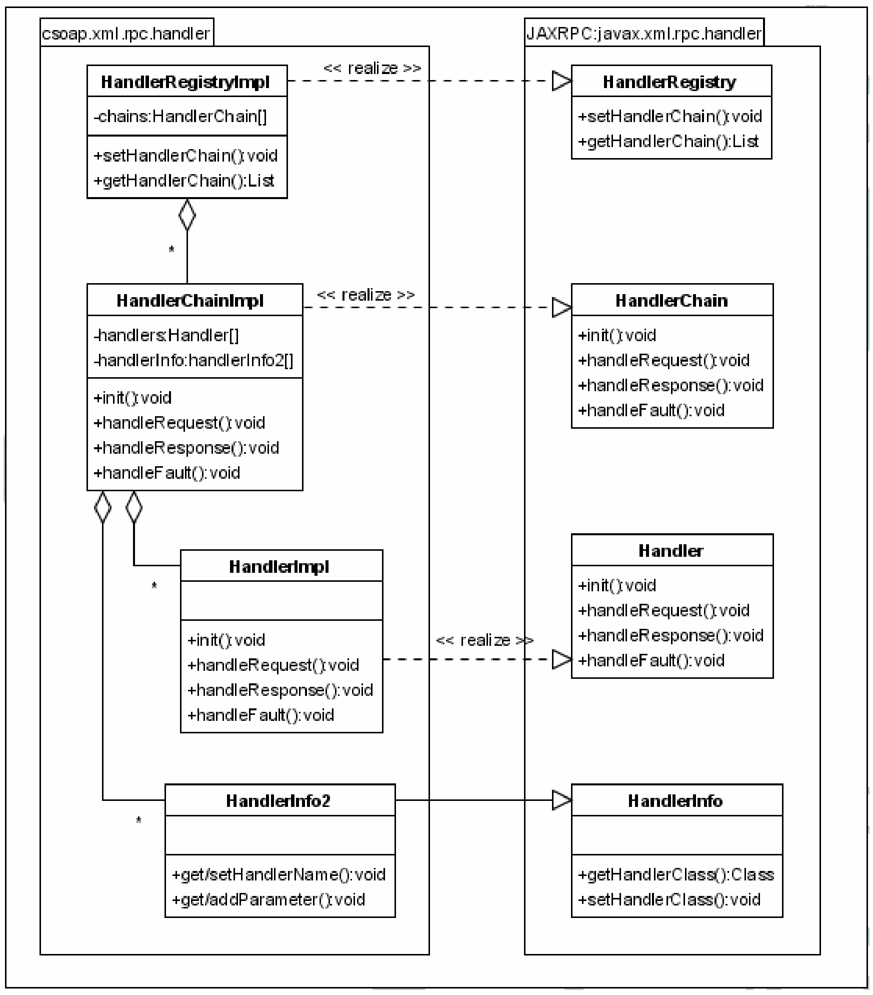
19. Sequence diagram of `ProvideManager` functioning(Service Session)

2.5. Handler subsystem

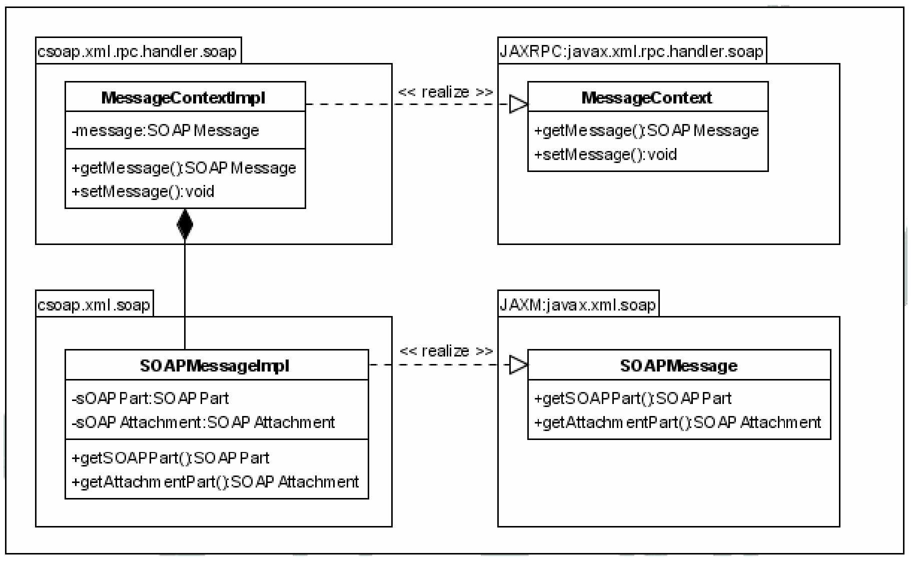
Sometimes we can need to process SOAP Messages before using them to invoke the Service (for example: security process, message tracking process) or before sending them (for example: compression process, redirection process…). CSOAP allows defining an Handler, that is a Java class which implements the `javax.xml.rpc.handler.Handler` interface and implements a processing algorithm over a SOAP message. The `HandlerChain` implements the `java.xml.rpc.handler.HandlerChain` and is a composite Handler, i.e. it aggregates a collection of Handlers. Handler subsystem takes care of the management of handlers and of their execution. For more details see the class diagram in Figure 20.

The Handler Chains are either Global or Service-specific. The Global Chain processes all messages which cross the CSOAP Engine, while the Service-specific Chain processes only the messages addressed to a specific Service. Each of these Chains consists of two Chains, a Request chain and a Response chain. The Request Chain processes the request messages and Response Chain processes the response messages. CSOAP let the user configure these Chains.

An object `SOAPMessageContext` representing the SOAP message goes through the Handler chains. As you can see in Figure 21, the `SOAPMessageContextImpl` implements the interface `javax.xml.rpc.handler.soap.SOAPMessageContext` and contains some properties of CSOAP Engine and a SOAP Message (implementing the `javax.xml.soap.SOAPMessage` interface) that can be either a request Message or a response Message.

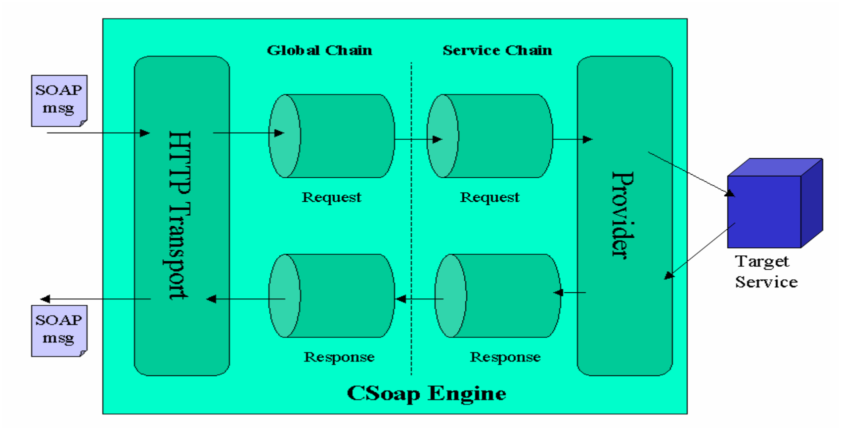


20. Class diagram of `HandlerChain`



21. Class diagram of `SOAPMessageContext`

The server-side message path is shown in Figure 22. The cylinders represent Handlers Chains (described above).



22. Message path on the CSOAP Engine Server.

The message arrives (in some protocol-specific manner) at a Transport Listener. In this case the Listener is a HTTP Servlet (is the unique listener provided by CSOAP). The Listener passes the message on the protocol-specific data into CSOAP Engine. The CSOAP engine's first job is to package the data into a Message object (`javax.xml.soap.Message`), and put the Message into a `SOAPMessageContext`. The `SOAPMessageContext` is also loaded with various properties by the CSOAP Engine.

Once the `SOAPMessageContext` object is ready, the CSOAP Engine passes it to the Global Request Chain. Each handler of the chain is invoked with `handleRequest()` method in request case and `handleResponse()` method in response case. After the Global request chain, the engine locates a Service request Chain, if configured, and then invokes any Handler specified therein.

Then, the Engine forwards the `SOAPMessageContext` object to provider subsystem. The provider will find all the necessary information to invoke the service such as, class name, method name, values of parameters.

The response returned by the service is put into a `SOAPMessageContext` object by the CSOAP Engine. The `SOAPMessageContext` object goes through the Service Response Handler Chain and the Global Response Handler Chain, and it is finally sent by the Transport Listener to the caller Client.

The Message Path on the client side is similar to that on the server side, except for the order of scoping that is reversed. The Service chain, if any, is called first - on the client side.

2.6. Deployment subsystem

This subsystem provides a way of deploying services into CSOAP given their description, which contains all the information related to the service (name, implementing class, methods and parameters, handler chains and type Mapping). CSOAP defines an XML based grammar for the service description, called WSDD (Web Service Deployment Descriptor). The description of all the deployed services is stored in a file named `wsdd.conf`. This file is read at each CSOAP server startup and it is updated when a service is deployed or undeployed.

The following is the structure of the WSDD grammar.

```
<wsdd>
 <deployment>
  <!-- handler declaration -->
  <handler name="handlerName" type="handlerClass">
    <parameter name="handlerParamName" value="paramValue"/>
  </handler>
  <service name="serviceName" scope="Request/Application/Session"
           provider="providerClass">
    <requestFlow>
      <!—list of handlers chains -->
      <handler type="handlerName">
    <requestFlow/>
    <responseFlow>
      <!—list of handlers chains -->
      <handler type="handlerName">
    <responseFlow/>
    <parameter name="className" value="java class name"/>
    <operation name="operationName" qname="operationXMLName"
               returnQname="XMLName" returnType="XMLType"/>
      <parameter name="parameterName" type="parameterType"
                 mode="IN/OUT/INOUT"></operation>
    <beanMapping qname="XMLType" type="javaClass"/>
    <typeMapping qname="XMLType" type="javaClass"
                 serializer="serializerFactoryClass"
                 deserializer="deserializerFactoryClass"
                 encodingStyle="encodingStyleName"/>
  </service>
 <deployment>
</wsdd>
```

- **`<service name="serviceName" scope="Request/Application/Session" provider="providerClass">`**

This tag allows defining the name of the service and the provider class. The default provider is `RCPProvider` class. The user can develop and use a new provider to invoke the deployed web services. The user must create a Java class that implements the `runService()` method and specify this class in provider attribute.

- **`<parameter name="className" value="java class name"/>`**

This tag defines the Java class that the `RPCProvider` must associate to the service. Any public method on that class may be called from any SOAP Client.

- **`<operation name="operationName" qname="operationXMLName" returnQname="XMLName" returnType="XMLType"/> </operation>`**

All the methods of the deployed service must be defined with the operation tag. This tag allows defining the operation name, the XML operation name, XML return value name, method's XML return type and all parameters of this method.

- **`<parameter name="parameter name" type="parameter type" mode="IN/OUT/INOUT">`**

For each operation you must define all the parameters. Each parameter is defined by a tag parameter which contains the parameter's name, parameter's type and parameter's mode (IN, OUT or INOUT).

- **`<Handler>`**

Defines a handler by giving the handler name and handler class.

- **`<requestFlow>`**

Indicates the list of Handlers that must be invoked when a call to this service is received, before the service invocation. Each handler must be defined before by **`<handler>`** tag.

- **`<responseFlow>`**

Indicates the list of Handlers that must be invoked when a call to this service is received, after service execution and before sending the SOAP Response message to the client.

A global handlers chain (*Request* and *Response*) is defined by using the "*" as a service name. All handlers defined in **`<requestFow>`** and **`<responseFlow>`** inside the **`<service name="*">`** are invoked when a deployed service is invoked.

- **`<beanMapping qname="XMLType type="javaClass" >`**

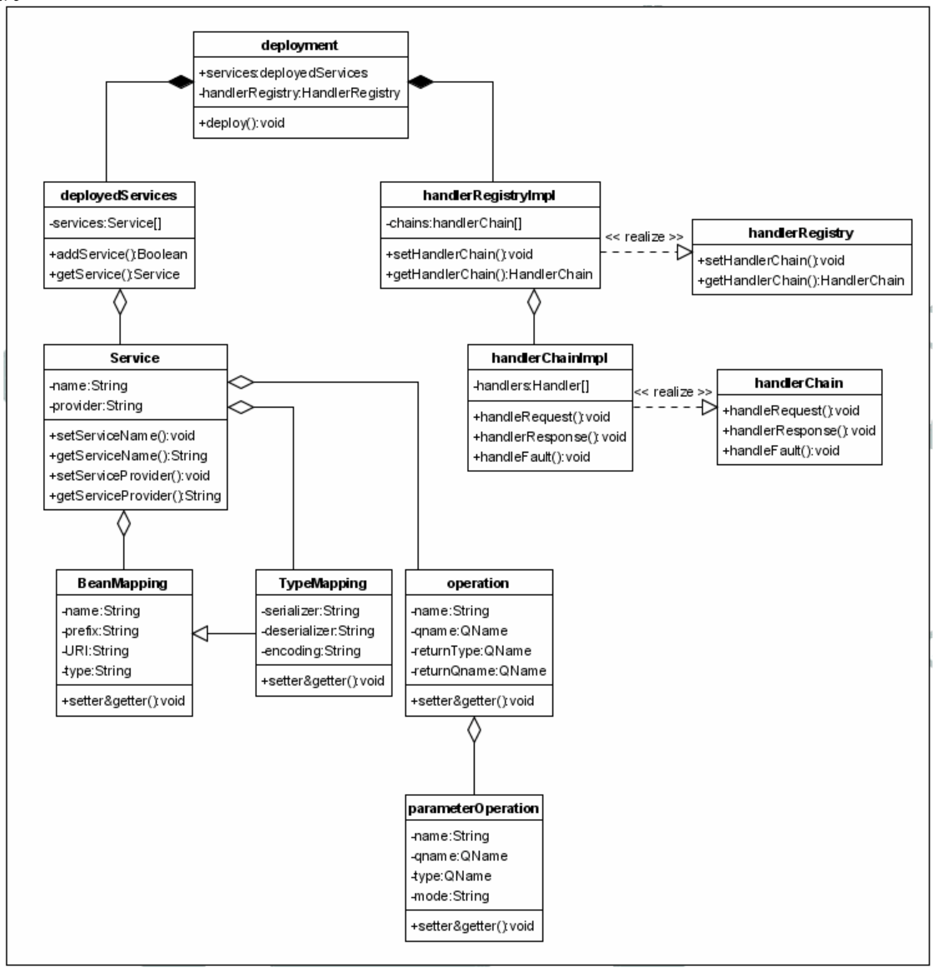
Defines a mapping between the specified `XMLType` and a `javaClass`.

- **`<typeMapping qname="XMLType" type="javaClass"`**
```
                 serializer="serializerFactoryClass"
                 deserializer="deserializerFactoryClass"
                 encodingStyle="encoding"/>
```

It's like the **`<beanMapping>`** tag, but there are three extra attributes:

- **`serializer`**, is the Java class name of the serializer factory which provides the serializer to be used to serialize an object of the specified Java class into XML.
- **`deserializer`**, is the class name of a deserializer factory that provides the deserializer to be used to deserialize XML into the correct Java class.
- **`encodingStyle`**, which is the used encoding.

This structure is mirrored also by a class hierarchy of factories for runtime artefacts. The following diagram (Figure 23) shows the classes and the types of runtime artefacts they produce.



23. Class diagram of deployment subsystem

- **`Deployment`**: reads the WSDD deployment files and adds the service descriptions, the Handler chains and the typemapping contained in the file to the memory data structures represented by instances of `deployedService`, `handlerRegistryImpl` et `typeMappingRegistryImpl` classes.
- **`DeployedService`**: contains an array of Service class instances and allows the handling of this structure with methods `addService()`, `getService()`, `removeService()`.
- **`Service`**: stores a service description.
- **`BeanMapping`**: stores a mapping between a Java type(`type`) and an XML type(`qname`).
- **`TypeMapping`**: this class is a subclass of `beanMapping` and defines some new attributes to store the serializer, the deserializer and the encoding that are used for mapping.
- **`Operation`**: represents a service method. It contains the name of the method, the XML name of the method, the return type, the list of parameters that is represented as an array of `parameterOperation`.
- **`ParameterOperation`**: contains the information about a method parameter: attribute name, type and mode (IN, OUT or INOUT).
- **`HandlerRegistryImpl`**: it's a *registry* of handler chains that implements the interface `javax.xml.rpc.handler.handlerRegistry`. The method `setHandlerChain()` allows configuring a new chain and associate it to a port. The method `gethandlerChain()` allows retrieving the configured chain by giving its port name. See section 2.5 for more details.
- **`HandlerChainImpl`**: it's an implementation of the interface `javax.xml.rpc.handler.handlerChain`. The method `handleRequest()` allow the

execution of the processing of messages SOAP Request. The method `handlerResponse()` allows the execution of the processing of messages SOAP Response and the method `handlerFault()` allow the execution of the processing of messages SOAP Fault. See Section 2.5 for more details.

CSOAP uses an XML parser based on SAX and compliant with JAXP (*Java API for XML Processing, see* `http://java.sun.com/xml/jaxp/index.jsp`) to read WSDD files. As already seen, using JAXP parsers involves implementing `org.xml.sax.helpers.DefaultHandler` interface (see section 2.2)
In this case, the interface `org.xml.sax.helpers.DefaultHandler` is implemented by the class `csoap.deployment.SAXHandler` that builds `csoap.deployment.Service` objects for each service entry inside the WSDD file.
The class `csoap.deployment.SAXHandler` makes use of two state transition tables. In Figure 24 you can see the table that is examined when an XML start element is found and the `startElement()` method of the `csoap.xml.soap.SAXHandler` is invoked. The state will be updated taking into account the content of the table and the right object will be created and added to the `Service` object. For example when the start of Envelope element is found, a `SOAPEnvelope` object is created and all its attributes and elements are added. End element state table in Figure 25 is examined at the end of each XML element, when the `endElement()` method of the `csoap.xml.soap.SAXHandler` is invoked.

Element Stat	deploym ent	Handler	Paramet er	Service	Request Flow	Respons eFlow	BeanMa pping	TypeMa pping	Wsdd	operatio n
0	1								0	
1		2		3						
2			4							
3			5		6	7	8	9		15
4										
5										
6		10								
7		12								
8										
9										
10										
11		5				7	8	9		
12										
13							8	9		15
14										
15			16							
16										

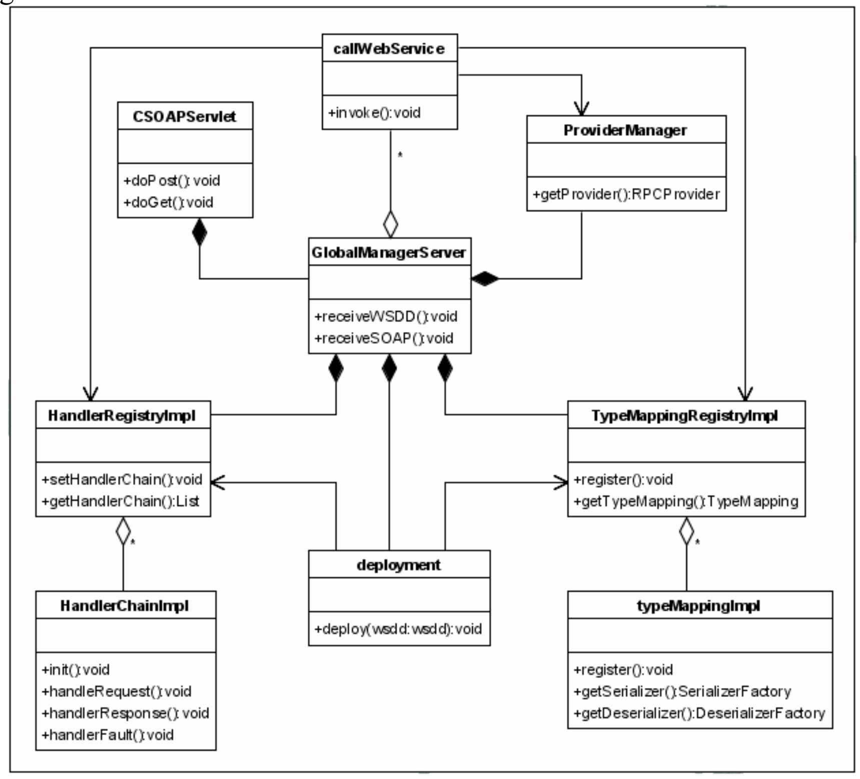
24. WSDD State transition table (start element)

Element Stat	deployment	Handler	Parameter	Service	Request Flow	ResponseFlow	BeanMapping	TypeMapping	Wsdd	operation
0										
1		2								
2		1		14						
3										
4			2							
5			13							
6										
7										
8							13			
9								13		
10										
11				14						
12										
13				14						
14	0									
15										3
16			15							

25. WSDD State transition table (start element)

2.7. Global Managing subsystem

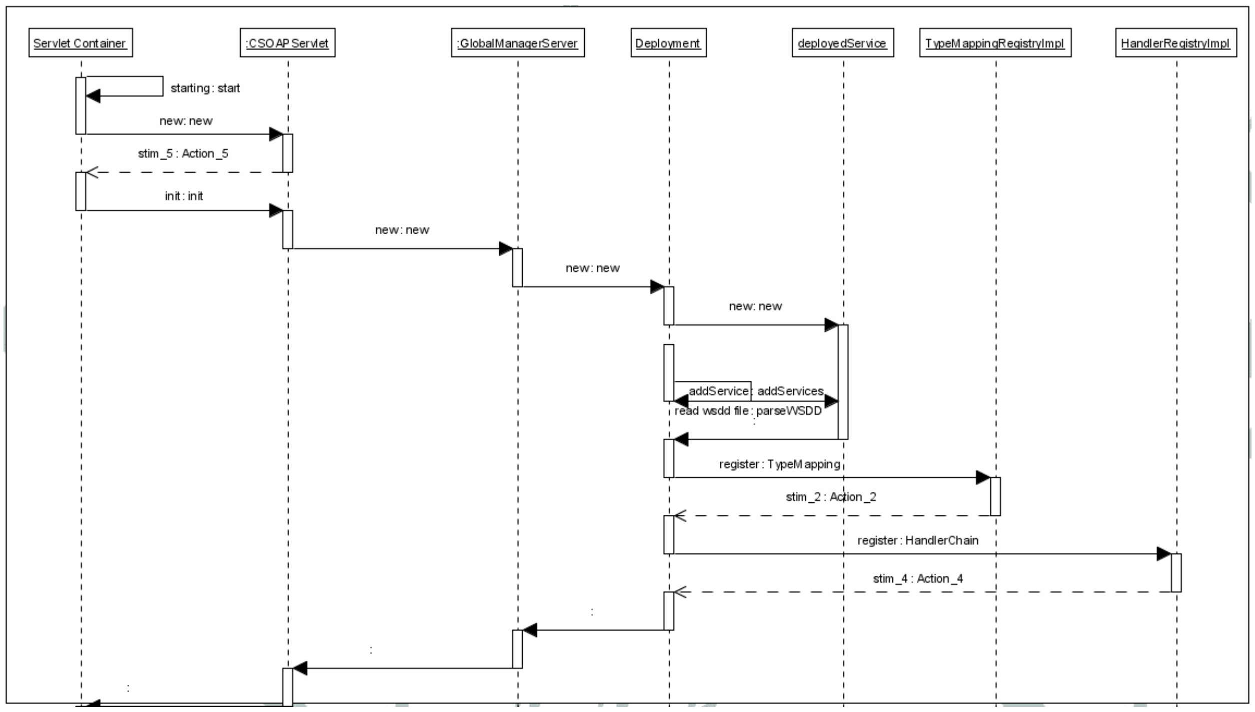
The Global Managing subsystem contains all server-side classes described below, and takes care of their coordination. The Global Managing subsystem receives input messages from the `CSOAPServlet` servlet. The following class diagram (Figure 26) shows the server-side classes and their relation-ships into Global Managing.



26. Class diagram of Global Managing subsystem

Startup

The following sequence diagram (Figure 27) shows the CSOAP Server startup phase. The CSOAP Server initialization consists in the creation of a static instance of class `GlobalMangerServer` that creates one instance of `deployedServices`, one of `TypeMappingRegistryImpl`, and one of `HandlerRegistryImpl`. It also creates an instance of the `deployment` class that parses the wsdd file (`wsdd.conf`), by using `csoap.deployment.SAXHandler` class (described in Section 2.6), and adds all the service objects returned by `SAXHandler` to `deployedServices`, registers the handler chains and the type mappings in `HandlerRegistryImpl` and `TypeMappingRegistryImpl`.

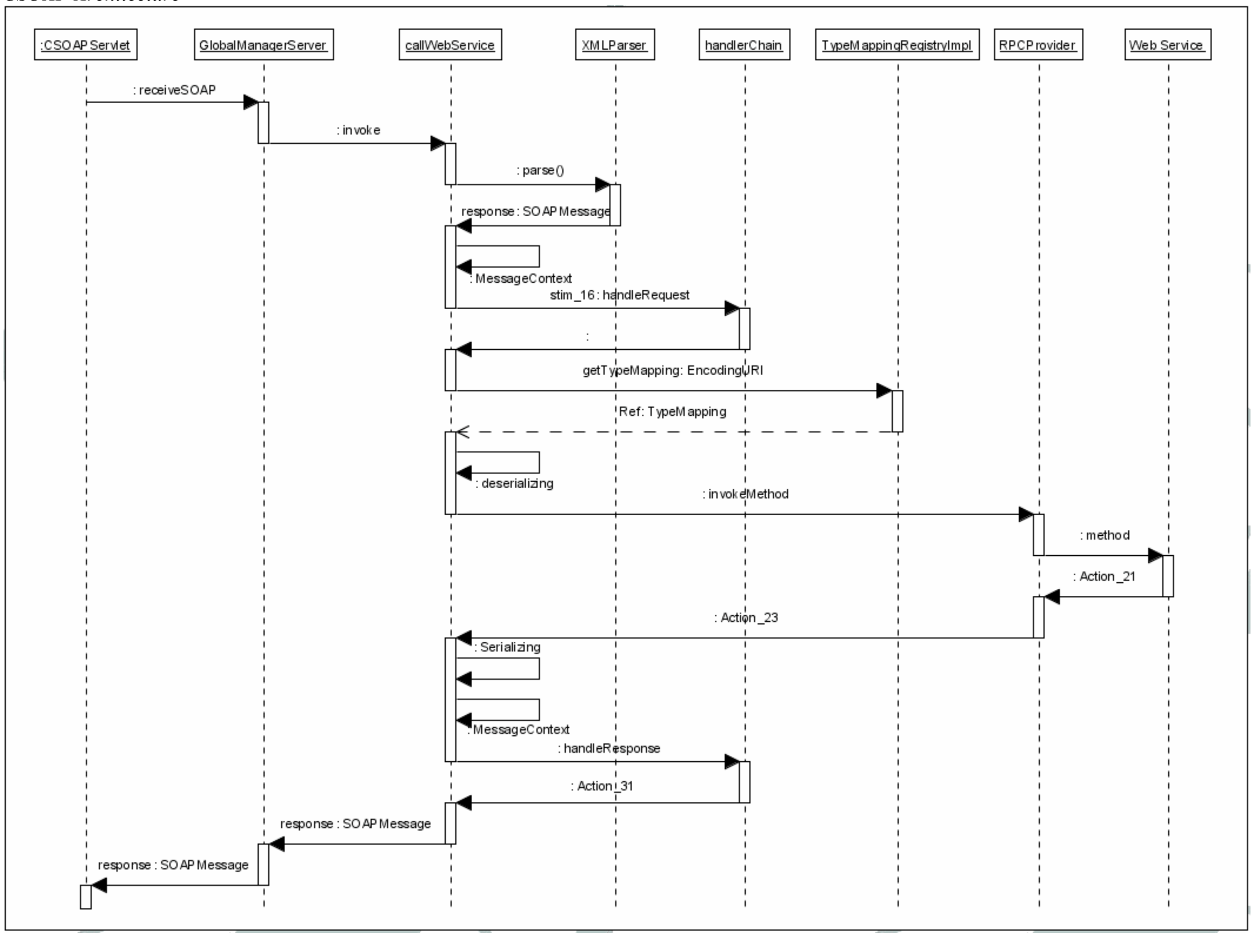


27. sequence diagram of CSOAP Server startup

Executing a Service Call on the Server Side

The sequence diagram in Figure 28 shows how CSOAP handles the reception of a SOAP request message. The `CSOAPServlet` waits for SOAP messages and when a SOAP message is received from a Client, the `CSOAPServlet` passes it to the `GlobalManagerServer` which creates a new instance of `callWebService` and gives it the received SOAP message as an `inputStream` object.

The `callWebService` uses the Model Message subsystem to model the SOAP message into a standard *SOAPMessage* object by using an event-based XML Parser, and it puts the *SOAPMessage* object into a `SOAPMessageContext`. This object is then processed by handlers (see Section 2.5 for more details) and then it is deserialized and passed to a `RPCProvider` instance (see Provider subsystem in section 2.4) which, use the serialized information contained in SOAP message to invoke the method of the target service and returns the result of the method invocation. The `callWebService` serializes this result into a standard `SOAPMessage` object (called Response SOAP Message) by using the `typeMapping` given by `TypeMappingRegistry` (see Section 2.3 for more detail) and puts the `SOAPMessage` into a `SOAPMessageContext`, which, after having been processed by handlers, is sent to the caller Client over HTTP.



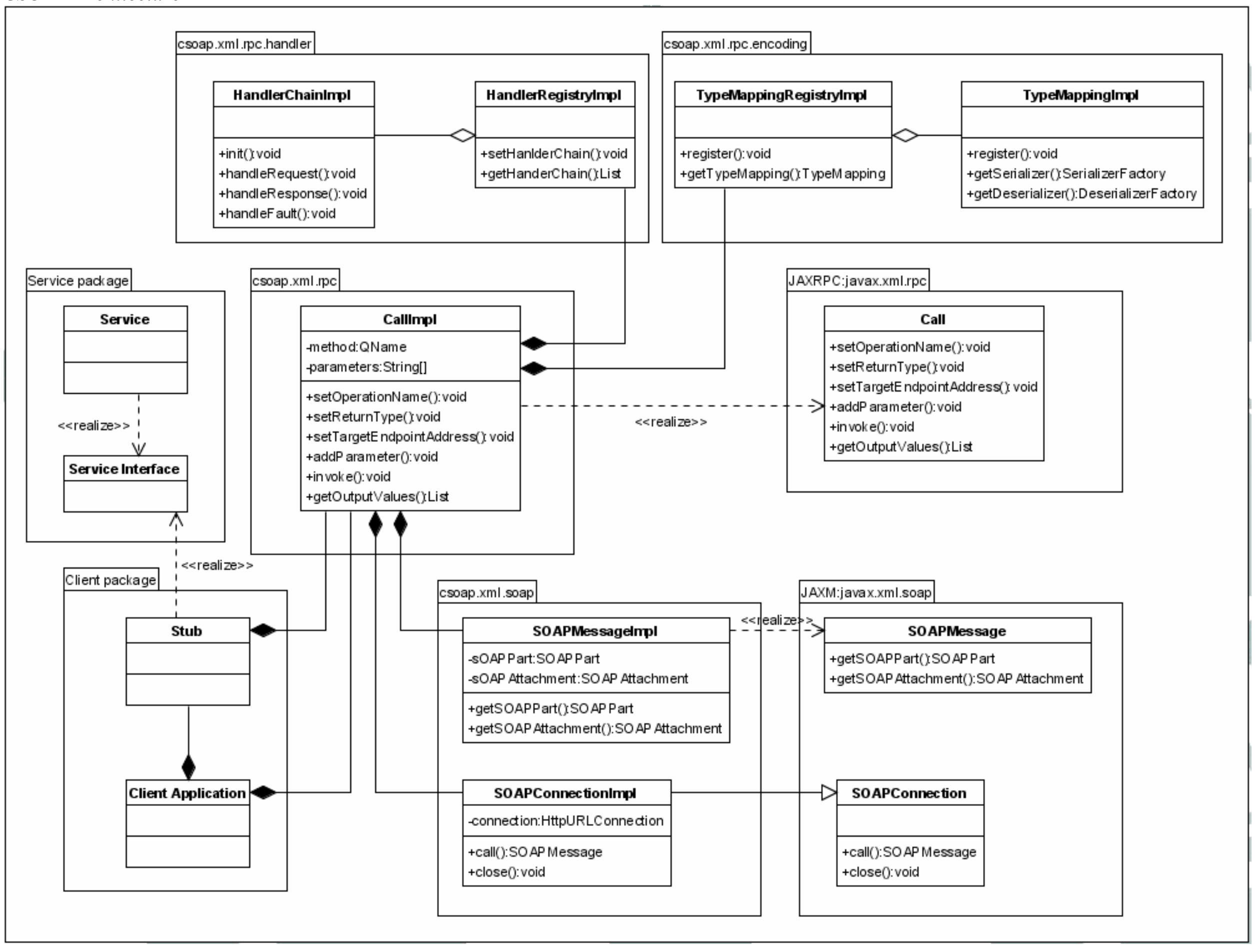
28. Sequence diagram of a service call on CSOAP Server side

2.8. CSOAP Client subsystem

In Figure 29 you can see a class diagram of the CSOAP Client subsystem.

As previously said, CSOAP Client has the role to make easier the development of client applications and to invoke web services. The main class in CSOAP Client is `CallImpl` class that implements interface `javax.xml.rpc.Call` and provides the methods required to configure a RPC call (method name, service address) and the `invoke()` that executes the service call.
`CallImpl` class contains an instance of class `SOAPMessage` that is used to model request and response messages. It also makes use of class `SOAPConnection` that sends SOAP request messages and receives SOAP response messages.

The developer can use the class `CallImpl` inside his application code or use the service Stub generated by CSOAP Generator (see Section 3) that provides the same interface as Service and that hides class `CallImpl` utilisation and configuration details.

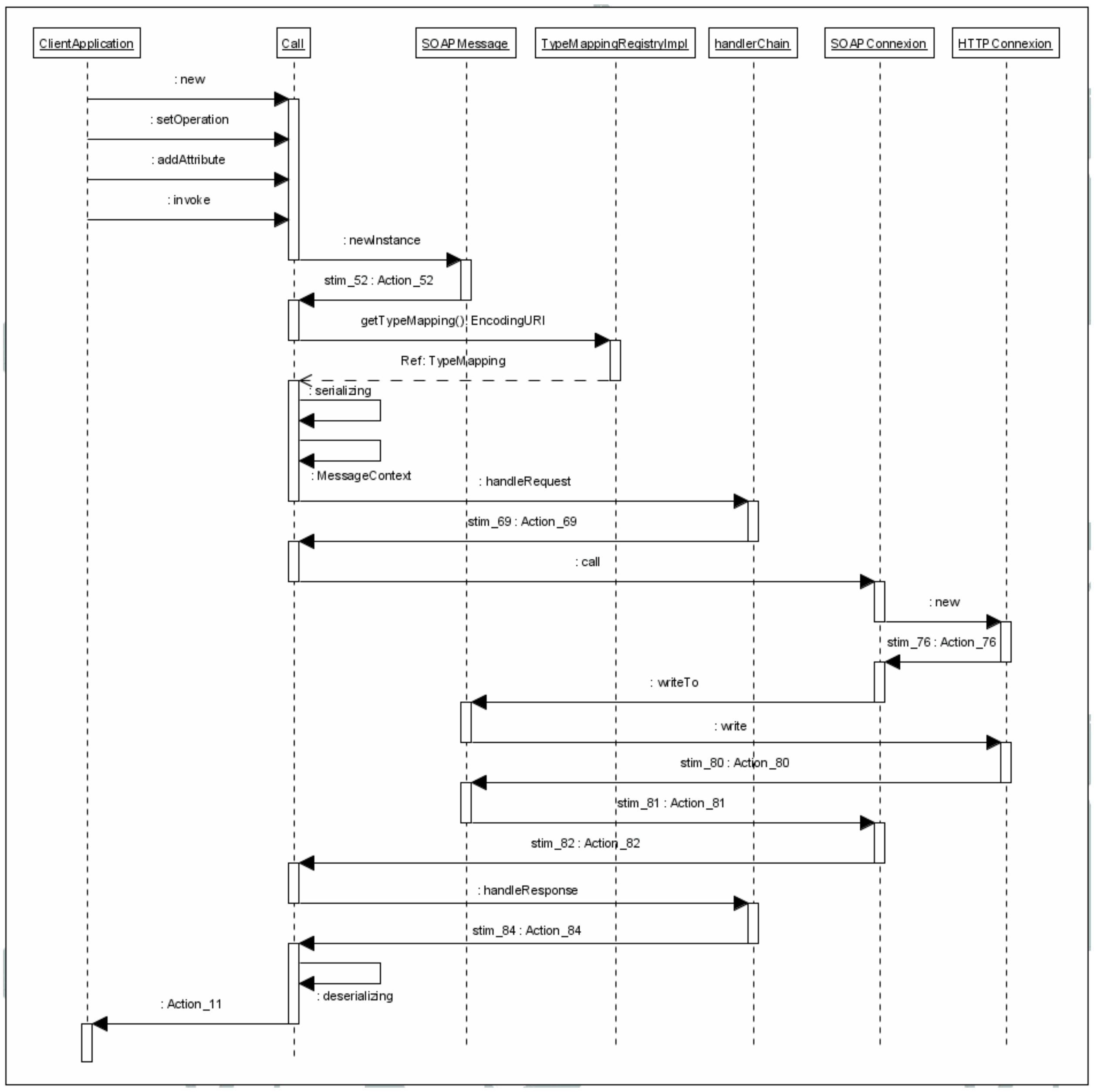


29. CSOAP Client class diagram

The sequence diagram in Figure 30 shows the execution of a service call on CSOAP client side.

To invoke a Web Service deployed in CSOAP Server the user Client Application creates a new instance of `CallImpl` which provides all necessary methods to define the service call. The data field to be initialized for a call are: the operation qualified name and its return type (if any), and the method parameter names and types.
When these field have been initialized, the invoke method can be called with the parameter values and the service call will be made. This method takes care of the creation of a new instance of standard `SOAPMessage` that is serialized taking into account all the message related information (operation name, parameters values, parameters types…) by using the `TypeMapping` given by the `TypeMappingRegistry` (see Section 2.3 for more detail).
The `SOAPMessage` is put into a `SOAPMessageContext` and goes through the handler chain. After handler processing (see Section 2.5 for more detail), the `Call` creates a new instance of `SOAPConnection` which provides a Call method that sends the given `SOAPMessage` to the given `endPoint` URL. The `SOAPConnection` creates a new instance of `HTTPConnection` with the given `endpoint` URL and invokes the `writeTo` method giving in input the `HTTPConnection` `outputStream`. The content of the `SOAPMessage` is written into the `outputStream` and sent to the `endpoint` URL.
When the remote server replies to the SOAP message, the received SOAP message result is put into `SOAPMessage` and `SOAPMessageContext` and goes through the handlers. Finally the `CallImpl`

deserializes the response `SOAPMessage` and returns the result of the invoke method to the Client Application as a Java object.



30. Sequence diagram of a service call on CSOAP Client side

3. CSOAP Generator

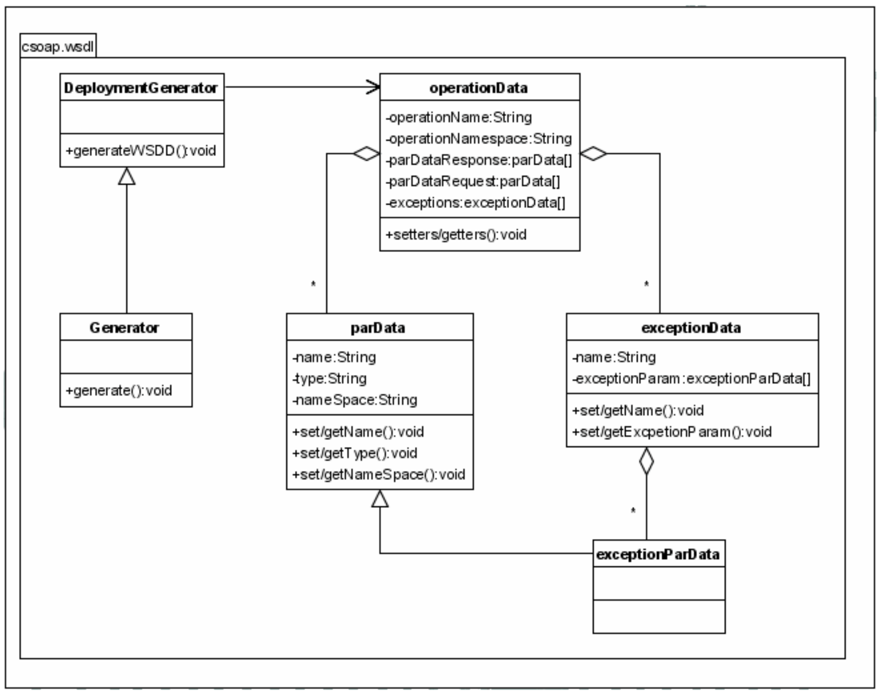
3.1. Introduction

The development of a Web service is a process that requires writing large pieces of code that follow in many cases the same patterns. This is more evident in the case of the development of the software components that are necessary for Web Service remote invocation. You have to write the stub and all the classes related to the data serialization process. For these reasons, we have developed a code generator that automatically generates a large part of client-side Java files that are involved in these mechanisms and the WSDD deployment file given its well-formed WSDL file. These files are generated according to CSOAP.

More specifically, with respect to Web services invocation, our component creates all the files that are necessary to invoke a service that is described by a standard WSDL document; in other words, it generates a stub class, an interface class, the related user-defined types (including holders for data serialization) and exceptions classes.

3.2. CSOAP Generator Architecture

The class diagram in Figure 31 shows the CSOAP Generator classes.



31. Class diagram of CSOAP Generator

The class `DeploymentGenerator` is responsible for the generation of wsdd files. `generateWSDD()` method allows the generation of a WSDD file from a WSDL file. The parameters for this method are:

- **`fileName`**: is the name of the WSDL file.
- **`side`**: indicate a server or client side (**s** for server side and **c** for client side).
- **`verbose`**: boolean value, if is *true* the debug information is displayed.
- **`dirName`**: is the name of the directory where the files are generated. A directory tree is generated according to the Java standard rules for classes and packages retrieving.
- **`implClass`**: is the name of a class that implements the service.
- **`scope:`** the value of service scoping (Request, Application or Session).

The class `Generator` extends the class `DeploymentGenerator` and provides the generation of classes for all the data types described in WSDL file and a service stub.

`generate()` method requires the same parameters as `generateWSDD()` method of `DeploymentGenerator` class.

The following table shows the files generated by the `generate()` method of CSOAP `Generator` class.

WSDL clause	**Java class(es) generated**
For each entry in the type section	A java class A holder if this type is used as an inout/out parameter
For each portType	A java interface
For each binding	A stub class
For each service	A service interface A service implementation One deploy.wsdd file with operation meta data One undeploy.wsdd file

3.3. WSDL elements and Java generated classes

Types

The names of the Java class generated from a WSDL file will be based on the type names definedin the WSDL file, for example, given the following WSDL:

```
<xsd:complexType name="hotelDescription">
  <xsd:sequence>
   <xsd:element name="hotelId" type="xsd:int"/>
   <xsd:element name="hotelName" type="xsd:string"/>
   <xsd:element name="hotelAddess" type="xsd:string"/>
   <xsd:element name="hotelPrice" type="xsd:long"/>
  </xsd:sequence>
</xsd:complexType>
```

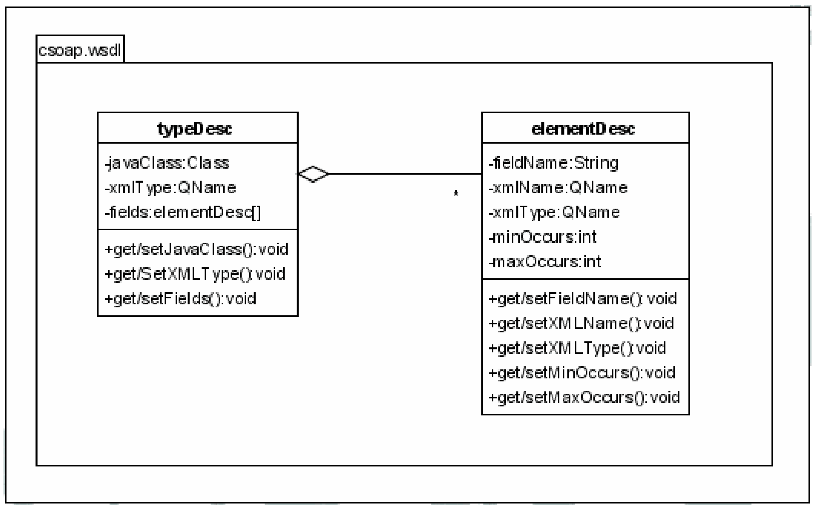
CSOAP Generator will generate:

```
public class Phone implements java.io.Serializable {

int hotelID;
String hotelName;
String hotelAddress;
String hotelPrice;

  public hotelDescription() {...}
  public int getHotelId() {...}
  public void setHotelId(int id) {...}
  public java.lang.String getHotelName() {...}
  public void setHotelName(java.lang.String name) {...}
  public java.lang.String getHotelAddess() {...}
  public void setHotelPrice(long price) {...}
  public long getHotelPrice() {...}
  public void setHotelAddress(java.lang.String address) {...}
  public boolean equals(Object obj) {...}
  public csoap.wsdl.typeDesc getTypeDesc() {...}
}
```

The generated class contains all the attributes defined in WSDL element `<complexType>` and the associated setter and getter methods.
`getTypeDesc()` method returns a `csoap.wsdl.typeDesc` object containing the generated class description. Class diagram in Figure 32 shows the class `typeDesc`.
That contains the type of the generated class (`javaClass`), the XML type associated to the Java class and the attribute list. Each attribute is represented by an `elementDesc` element of the `fields` array.
`elementDesc` class contains all the information describing a class attribute (field name, XML name, XML type, minimum occurrences, maximum occurrences).
`typeDesc` class is used by CSOAP to serivalize and deserialize object instances and complex classes.



***32.* Class diagram of `TypeDesc`**

Arrays declaration

According to WSDL specification, arrays must be declared by restriction. In other words, if you have to declare an array of Java String, you define it as an element belonging to a subset (a restriction) of the set of standard SOAP arrays. More formally, an array is derived from a `soapenc:array` by restriction using the `wsdl:arrayType` attribute, with the soapenc prefix associated to the namespace URI: `http://schemas.xmlsoap.org/soap/encoding` and the wsdl prefix to `http://schemas.xmlsoap.org/wsdl`.

The following example shows the definition of an array of String derived from the `soapenc:array` by restriction:

```
<xsd:complexType name="ArrayOfHotelDescription">
  <xsd:complexContent>
    <xsd:restriction base="soapenc:Array">
      <xsd:sequence>
        <xsd:element name="hotelDescription" minOccurs="0"
        maxOccurs="unbounded" type="tns:HotelDescription"/>
      </xsd:sequence>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

It is worth noting that the name of the complex type must start with `ArrayOf`; this limitation derives from the implementation of the WSIF library that we have used to manipulate the XML Schema declaration inside the WSDL file.

Holders

This type may be used as an *inout* or *out* parameter. Java does not have the concept of *inout/out* parameters. In order to achieve this behavior, JAX-RPC specifies the use of holder classes. A holder class is simply a class that contains an instance of its type. For example, the holder for the `hotelDescription` class would be

```
package csoap.examples.hotelService.holders;
public final class HotelDescriptionHolder implements
javax.xml.rpc.holders.Holder {
 public csoap.examples.hotelService.HotelDescription value;

 public HotelDescription Holder()
 {
 }

 public HotelDescription
Holder(csoap.examples.hotelService.HotelDescription value) {
  this.value = value;
 }
}
```

A holder class is only generated for a type if that type is used as an inout or out parameter. Note that the holder class has the suffix "Holder" appended to the class name, and it is generated in a sub-package with the "holders".
The holder classes for the primitive types can be found in `javax.xml.rpc.holders`.

PortTypes

The Service Definition Interface (SDI) is the interface that's derived from a WSDL's `portType`. This is the interface you use to access the operations on the service. For example, given the WSDL

```
<message name="listHotelsRequest">
  <part name="region" type="xsd:string"/>
</message>

<message name="listHotelsResponse">
<part name="hotels" type="tns:ArrayOfHotelDescription"/>
</message>

<portType name="HotelInterface">
  <operation name="listHotles">
    <input message="tns:listHotelsRequest"/>
    <output message="tns:listHotelResponse"/>
  </operation>
</portType>
```

CSOAP Generator will generate

```
public interface HotelInterface extends java.rmi.Remote {
 public                 csoap.examples.hotelService.HotelDescription[]
listHotels(String region) throws
  java.rmi.RemoteException;
}
```

A note about the name of the SDI. The name of the SDI is typically the name of the `portType`. However, to construct the SDI, CSOAP Generator needs information from both the `portType` and the binding. (This is unfortunate and is a topic of discussion for WSDL version 2.)
JAX-RPC says: "The name of the Java interface is mapped from the name attribute of the `wsdl:portType` element. If the mapping to a service definition interface uses elements of the `wsdl:binding`, then the name of the service definition interface is mapped from the name of the `wsdl:binding` element."
Note the name of the spec. It contains the string "RPC". So this spec, and CSOAP Generator, assumes that the interface generated from the `portType` is an RPC interface.

Bindings

A Stub class implements the SDI. Its name is the binding name with the suffix "Stub". It contains the code which turns the method invocations into SOAP calls using the CSOAP Service and Call objects. The Stub letting you call the Service exactly as if it were a local object. In other words, you don't need to deal with the endpoint URL, namespace, or parameter arrays which are involved in dynamic invocation via the Service and Call objects. The Stub hides all that work for you.
Given the following WSDL:

```
<binding name="HotelSOAPBinding" type="tns:HotelInterface">
  ...
</binding>
```

CSOAP Generator will generate:

```
public class HotelSOAPBindingStub implements
HotelService.HotelInterface {
{
 public HotelSOAPBindingStub(URL endpointURL,
          javax.xml.rpc.Service service)
 {...}

public HotelService.Hotel[] listHotels(java.lang.String city) throws
java.rmi.RemoteException,HotelService.HotelException {
// implementation of Call listHotels
// return the result of invokation
}
}
```

Exceptions

You should pay particular attention in declaring exceptions; in fact, you must follow the standard defined in JAX-RPC specification. Now we focussed in particular on service specific exceptions. To sum up, a fault element, which is an optional element inside an operation block, specifies the format of any error message related to the remote invocation of a particular service method. It is worth noting that according to the WSDL specification, a fault message must have a single part.

The name of the Java exception is mapped from the name attribute of the message. For reference, you can look at the following example:

```
<message name="HotelException">
  <part name="errorMessage" type="xsd:string"/>
</message>
...
<portType name="HotelInterface">
  ...
  <operation name="listHotels">
    <input>...</input>
    <output>...</output>
    <fault name="HotelException" message="HotelException"/>
  </operation>
  ...
</portType>
```

The following is a fragment of the Java service endpoint interface derived from the above WSDL port type definition:

```
public interface HotelInterface extend java.rmi.Remote {
   public csoap.examples.hotelService.HotelDescription(String region)
      throws java.rmi.RemoteException,
            HotelService.HotelException;
}
```

In this example you can note that the fault element is mapped to the `com.HotelService.HotelException` exception that extends the `java.lang.Exception` class.

In this case the generator produces the following code for the definition of the user-defined exception class:

```
public class HotelException extends java.lang.Exception {
  public HotelException(java.lang.String errorMessage) {...}
  public getErrorMessage(){...}
}
```

It is worth observing that the single message part in the fault maps to a field with a getter method in the mapped exception. Moreover, it is declared as input parameter in the exception constructor.

4. Appendix

Sun's JAXM Specification

public interface **`SOAPElement`**
extends `Node`, `Element`
An object representing an element of a SOAP message that is allowed but not specifically prescribed by a SOAP specification. This interface serves as the base interface for those objects that are specifically prescribed by a SOAP specification.
Methods in this interface that are required to return SAAJ specific objects may "silently" replace nodes in the tree as required to successfully return objects of the correct type. See `getChildElements()` and `javax.xml.soap` for details.

public interface **`SOAPEnvelope`**
extends `SOAPElement`
The container for the SOAPHeader and SOAPBody portions of a `SOAPPart` object. By default, a `SOAPMessage` object is created with a `SOAPPart` object that has a `SOAPEnvelope` object. The `SOAPEnvelope` object by default has an empty `SOAPBody` object and an empty `SOAPHeader` object. The `SOAPBody` object is required, and the `SOAPHeader` object, though optional, is used in the majority of cases. If the `SOAPHeader` object is not needed, it can be deleted, which is shown later.
A client can access the `SOAPHeader` and `SOAPBody` objects by calling the methods `SOAPEnvelope.getHeader` and `SOAPEnvelope.getBody`. The following lines of code use these two methods after starting with the `SOAPMessage` object message to get the `SOAPPart` object *sp*, which is then used to get the `SOAPEnvelope` object *se*.

```
      SOAPPart sp = message.getSOAPPart();
      SOAPEnvelope se = sp.getEnvelope();
      SOAPHeader sh = se.getHeader();
      SOAPBody sb = se.getBody();
```

It is possible to change the body or header of a `SOAPEnvelope` object by retrieving the current one, deleting it, and then adding a new body or header. The `javax.xml.soap.Node` method `deleteNode` deletes the XML element (node) on which it is called. For example, the following line of code deletes the `SOAPBody` object that is retrieved by the method `getBody`.

```
       se.getBody().detachNode();
```

To create a `SOAPHeader` object to replace the one that was removed, a client uses the method `SOAPEnvelope.addHeader`, which creates a new header and adds it to the `SOAPEnvelope` object. Similarly, the method `addBody` creates a new `SOAPBody` object and adds it to the `SOAPEnvelope` object. The following code fragment retrieves the current header, removes it, and adds a new one. Then it retrieves the current body, removes it, and adds a new one.

```
      SOAPPart sp = message.getSOAPPart();
      SOAPEnvelope se = sp.getEnvelope();
      se.getHeader().detachNode();
      SOAPHeader sh = se.addHeader();
      se.getBody().detachNode();
      SOAPBody sb = se.addBody();
```

It is an error to add a `SOAPBody` or `SOAPHeader` object if one already exists.
The `SOAPEnvelope` interface provides three methods for creating `Name` objects. One method creates `Name` objects with a local name, a namespace prefix, and a namesapce URI. The second method creates

`Name` objects with a local name and a namespace prefix, and the third creates `Name` objects with just a local name. The following line of code, in which *se* is a `SOAPEnvelope` object, creates a new `Name` object with all three.

```
     Name name = se.createName("GetLastTradePrice", "WOMBAT",
                                  "http://www.wombat.org/trader");
```

public interface **`SOAPHeader`**
extends `SOAPElement`
A representation of the SOAP header element. A SOAP header element consists of XML data that affects the way the application-specific content is processed by the message provider. For example, transaction semantics, authentication information, and so on, can be specified as the content of a `SOAPHeader` object.
A `SOAPEnvelope` object contains an empty `SOAPHeader` object by default. If the `SOAPHeader` object, which is optional, is not needed, it can be retrieved and deleted with the following line of code. The variable *se* is a `SOAPEnvelope` object.

```
      se.getHeader().detachNode();
```

A `SOAPHeader` object is created with the `SOAPEnvelope` method `addHeader`. This method, which creates a new header and adds it to the envelope, may be called only after the existing header has been removed.

```
      se.getHeader().detachNode();
      SOAPHeader sh = se.addHeader();
```

A `SOAPHeader` object can have only `SOAPHeaderElement` objects as its immediate children. The method `addHeaderElement` creates a new `HeaderElement` object and adds it to the `SOAPHeader` object. In the following line of code, the argument to the method `addHeaderElement` is a `Name` object that is the name for the new `HeaderElement` object.

```
      SOAPHeaderElement shElement = sh.addHeaderElement(name);
```

public interface **`SOAPHeaderElement`**
extends `SOAPElement`
An object representing the contents in the SOAP header part of the SOAP envelope. The immediate children of a `SOAPHeader` object can be represented only as `SOAPHeaderElement` objects.
A `SOAPHeaderElement` object can have other `SOAPElement` objects as its children.

public interface **`SOAPFault`**
extends `SOAPBodyElement`
An element in the `SOAPBody` object that contains error and/or status information. This information may relate to errors in the `SOAPMessage` object or to problems that are not related to the content in the message itself. Problems not related to the message itself are generally errors in processing, such as the inability to communicate with an upstream server.
The `SOAPFault` interface provides methods for retrieving the information contained in a `SOAPFault` object and for setting the fault code, the fault actor, and a string describing the fault. A fault code is one of the codes defined in the SOAP 1.1 specification that describe the fault. An actor is an intermediate recipient to whom a message was routed. The message path may include one or more actors, or, if no actors are specified, the message goes only to the default actor, which is the final intended recipient.

public interface **`SOAPBody`**
extends `SOAPElement`
An object that represents the contents of the SOAP body element in a SOAP message. A SOAP body element consists of XML data that affects the way the application-specific content is processed.
A `SOAPBody` object contains `SOAPBodyElement` objects, which have the content for the SOAP body. A `SOAPFault` object, which carries status and/or error information, is an example of a `SOAPBodyElement` object.

public interface **`Detail`**
extends `SOAPFaultElement`
A container for `DetailEntry` objects. `DetailEntry` objects give detailed error information that is application-specific and related to the `SOAPBody` object that contains it.
A `Detail` object, which is part of a `SOAPFault` object, can be retrieved using the method `SOAPFault.getDetail`. The `Detail` interface provides two methods. One creates a new `DetailEntry` object and also automatically adds it to the `Detail` object. The second method gets a list of the `DetailEntry` objects contained in a `Detail` object.
The following code fragment, in which *sf* is a `SOAPFault` object, gets its `Detail` object (*d*), adds a new `DetailEntry` object to *d*, and then gets a list of all the `DetailEntry` objects in *d*. The code also creates a `Name` object to pass to the method `addDetailEntry`. The variable *se*, used to create the `Name` object, is a `SOAPEnvelope` object.

```
    Detail d = sf.getDetail();
    Name name = se.createName("GetLastTradePrice", "WOMBAT",
                              "http://www.wombat.org/trader");
    d.addDetailEntry(name);
    Iterator it = d.getDetailEntries();
```

public interface **`DetailEntry`**
extends `SOAPElement`
The content for a `Detail` object, giving details for a `SOAPFault` object. A `DetailEntry` object, which carries information about errors related to the `SOAPBody` object that contains it, is application-specific.

public abstract class **`SOAPMessage`**
extends `Object`
The root class for all SOAP messages. As transmitted on the "wire", a SOAP message is an XML document or a MIME message whose first body part is an XML/SOAP document.
A `SOAPMessage` object consists of a SOAP part and optionally one or more attachment parts. The SOAP part for a `SOAPMessage` object is a `SOAPPart` object, which contains information used for message routing and identification, and which can contain application-specific content. All data in the SOAP Part of a message must be in XML format.
A new `SOAPMessage` object contains the following by default:

- A `SOAPPart` object
- A `SOAPEnvelope` object
- A `SOAPBody` object
- A `SOAPHeader` object

The SOAP part of a message can be retrieved by calling the method `SOAPMessage.getSOAPPart()`. The `SOAPEnvelope` object is retrieved from the `SOAPPart` object, and the `SOAPEnvelope` object is used to retrieve the `SOAPBody` and `SOAPHeader` objects.

```
      SOAPPart sp = message.getSOAPPart();
      SOAPEnvelope se = sp.getEnvelope();
      SOAPBody sb = se.getBody();
      SOAPHeader sh = se.getHeader();
```

In addition to the mandatory `SOAPPart` object, a `SOAPMessage` object may contain zero or more `AttachmentPart` objects, each of which contains application-specific data. The `SOAPMessage` interface provides methods for creating `AttachmentPart` objects and also for adding them to a `SOAPMessage` object. A party that has received a `SOAPMessage` object can examine its contents by retrieving individual attachment parts.

Unlike the rest of a SOAP message, an attachment is not required to be in XML format and can therefore be anything from simple text to an image file. Consequently, any message content that is not in XML format must be in an `AttachmentPart` object.

A `MessageFactory` object may create `SOAPMessage` objects with behavior that is specialized to a particular implementation or application of SAAJ. For instance, a `MessageFactory` object may produce `SOAPMessage` objects that conform to a particular Profile such as ebXML. In this case a `MessageFactory` object might produce `SOAPMessage` objects that are initialized with ebXML headers.

In order to ensure backward source compatibility, methods that are added to this class after version 1.1 of the SAAJ specification are all concrete instead of abstract and they all have default implementations. Unless otherwise noted in the JavaDocs for those methods the default implementations simply throw an `UnsupportedOperationException` and the SAAJ implementation code must override them with methods that provide the specified behavior. Legacy client code does not have this restriction, however, so long as there is no claim made that it conforms to some later version of the specification than it was originally written for. A legacy class that extends the `SOAPMessage` class can be compiled and/or run against succeeding versions of the SAAJ API without modification. If such a class was correctly implemented then it will continue to behave correctly relative the the version of the specification against which it was written.

Sun's JAX-RPC Specification

public interface **`Deserializer`**
extends `Serializable`

The `javax.xml.rpc.encoding.Deserializer` interface defines a base interface for deserializers. A Deserializer converts an XML representation to a Java object using a specific XML processing mechanism and based on the specified type mapping and encoding style.

public interface **`DeserializerFactory`**
extends `Serializable`

The `javax.xml.rpc.encoding.DeserializerFactory` is a factory of deserializers. A `DeserializerFactory` is registered with a `TypeMapping` instance as part of the `TypeMappingRegistry.`

public interface **`Serializer`**
extends Serializable

The javax.xml.rpc.encoding.Serializer interface defines the base interface for serializers. A Serializer converts a Java object to an XML representation using a specific XML processing mechanism and based on the specified type mapping and encoding style.

public interface **`SerializerFactory`**
extends `Serializable`
The `javax.xml.rpc.encoding.SerializerFactory` is a factory of the serializers. A `SerializerFactory` is registered with a `TypeMapping` object as part of the `TypeMappingRegistry`.

public interface **`TypeMapping`**
The `javax.xml.rpc.encoding.TypeMapping` is the base interface for the representation of a type mapping. A `TypeMapping` implementation class may support one or more encoding styles.
For its supported encoding styles, a TypeMapping instance maintains a set of tuples of the type {Java type, `SerializerFactory`, `DeserializerFactory`, XML type}.

public interface **`TypeMappingRegistry`**
extends `Serializable`
The interface `javax.xml.rpc.encoding.TypeMappingRegistry` defines a registry of `TypeMapping` instances for various encoding styles.

public interface **`Call`**
The `javax.xml.rpc.Call` interface provides support for the dynamic invocation of a service endpoint. The `javax.xml.rpc.Service` interface acts as a factory for the creation of `Call` instances.
Once a `Call` instance is created, various setter and getter methods may be used to configure this `Call` instance

public interface **SOAPMessageContext**
extends MessageContext
The interface `javax.xml.rpc.soap.SOAPMessageContext` provides access to the SOAP message for either RPC request or response. The `javax.xml.soap.SOAPMessage` specifies the standard Java API for the representation of a SOAP 1.1 message with attachments.

public interface **Handler**
The `javax.xml.rpc.handler.Handler` interface is required to be implemented by a SOAP message handler. The `handleRequest`, `handleResponse` and `handleFault` methods for a SOAP message handler get access to the `SOAPMessage` from the `SOAPMessageContext`. The implementation of these methods can modify the `SOAPMessage` including the headers and body elements.

public interface **HandlerChain**
extends List

The `javax.xml.rpc.handler.HandlerChain` represents a list of handlers. All elements in the `HandlerChain` are of the type `javax.xml.rpc.handler.Handler`.
An implementation class for the `HandlerChain` interface abstracts the policy and mechanism for the invocation of the registered handlers.

public interface **HandlerRegistry**
extends Serializable
The `javax.xml.rpc.handler.HandlerRegistry` provides support for the programmatic configuration of handlers in a `HandlerRegistry`.
A handler chain is registered per service endpoint, as indicated by the qualified name of a port. The `getHandlerChain` returns the handler chain (as a `java.util.List`) for the specified service endpoint. The returned handler chain is configured using the `java.util.List` interface. Each element in this list is required to be of the Java type `javax.xml.rpc.handler.HandlerInfo`.

public class **HandlerInfo**
extends Object
implements Serializable
The `javax.xml.rpc.handler.HandlerInfo` represents information about a handler in the HandlerChain. A HandlerInfo Instance is passed in the `Handler.init` method to initialize a `Handler` instance.