*Process MeNtOR 3.o*

*Uni-SEP*

Web Services Extension to SIP

ECE355 – Software Engineering Fall 2005

**Design Document**

|  |  |
| --- | --- |
| Version: | 1.0 |
| Print Date: |  |
| Release Date: | October 21, 2005 |
| Release State: | Final |
| Approval State: | Approved |
| Approved by: |  |
| Prepared by: | Group #G012  NAMES HERE |
| Reviewed by: |  |
| Path Name: |  |
| File Name: | ECE355SDD.doc |
| Document No: |  |

# Document Change Control

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Authors** | **Summary of Changes** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Document Sign-Off

|  |  |  |
| --- | --- | --- |
| **Name (Position)** | **Signature** | **Date** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Contents

1 Introduction 4

1.1 Overview 4

1.2 Resources - References 5

2 Major Design Decisions 6

3 Architecture 7

4 Detailed Class Diagrams 11

4.1 UML Class Diagrams 11

4.2 Method Details 18

5 State Diagrams 20

6 Open Issues 21

7 Domain Dictionary 22

7.1 Terms and Abbreviations 22

# Introduction

## Overview

The programming tasks for this project are summarized as follows:

1. To modify the Location server to be able to query the Service Registry, and pass the list of services or NULL to the Proxy for further processing.
2. To modify the proxy server to be able to accept service descriptions from the registry and to compose MESSAGE BODY strings.
3. To modify the Client Agent to reference a Wrapper.
4. To create a Wrapper class that references the Client Agent. This Wrapper class should use the JAX-RPC API to invoke the Web Service that has been selected using the extended Client Agent UI.
5. To modify the template service and the wsdl specification file that will be given to create variant Web Services.

The SDD document contains the following information:

1. Component Diagram of the system (Architecture). The level of detail and granularity will be at the Java Package level of detail
2. Deployment Diagram of the system assuming current run-time configuration.
3. Detailed class diagram for all classes created or modified in the system with only one level of associations. The detailed class diagram contains the classes in UML notation and a table for each class with its data members and methods with the appropriate signatures. Design pattern are identified.
4. Pseudo code for all major methods in the classes written or modified. The major methods are the ones in the Proxy/Location server and, the SIP Communicator wrapper. These are the methods that initiate the request to the Location server/registry (Proxy server), querying the repository (Location server), building the message to be returned to the agent (Proxy), processing the message from the proxy (SIP Agent), calling the service (Wrapper).
5. State Diagram for the ***Call*** class.

## Resources - References

This SDD follows the design specification specified in

ECE355-2005-Project-Design-Description-v2

References for different technologies used in the project are listed below:

Eclipse: <http://www.eclipse.org/downloads/index.php>

WSDP: <http://java.sun.com/webservices/jwsdp/index.jsp>

JAXR: <http://java.sun.com/webservices/jaxr/index.jsp>

JAXRPC: <http://java.sun.com/webservices/jaxrpc/index.jsp>

WSIF: <http://ws.apache.org/wsif/>

JAIN-SIP: <https://jain-sip.dev.java.net/>

SIP-COMMUNICATOR: <https://sip-communicator.dev.java.net/>

ebXML (preferred Registry for our project): <http://www.ebxml.org/>

UDDI: <http://www.uddi.org/>

WSDL: <http://www.w3schools.com/wsdl/default.asp>

Java System Application Server: <http://java.sun.com/j2ee/1.4/download.html#sdk>

AXIS: <http://ws.apache.org/axis/>

SAMPLE CODE LOCATION : [http://www.swen.uwaterloo.ca/~kostas/ECE355-05/project/ece355-project- package.zip](http://www.swen.uwaterloo.ca/~kostas/ECE355-05/project/ece355-project-%09package.zip)

# Major Design Decisions

Most of the software code has been provided. Only two classes ***Proxy*** and ***WSEvent*** needs to be implemented to complete the operation of the Web Service Extension to SIP.

However there are assumptions that were made regarding the operation of this software:

1. The parser in the ***WSEvent*** takes a string of the form *<Number-of-services:Integer> -- <Service1-description:String> <Service1-Wsdluri::URL> <Service1-EndPoint> -- <Service2-description:String> <Service2-Wsdluri::URL> <Service2-EndPoint>* and it is assumed that no where in the service description, wsdl URI, or endpoint fields that there will be a control character of the form “ --“ that the parser needs to parse the string correctly to create a ***WSListData*** object. Also the wsdl URI and endpoints are assumed not to contain any spaces. The parser uses the ***String.split()*** method.
2. The callee agent must be registered to the proxy server for the Web Services invocation to function. Thus if the callee agent must be online otherwise the execution will not get to the point where the Service Registry will be queried.

The sample code provided contained several design patterns:

1. The Web Service invocation uses a Strategy design pattern in choosing what kind of Web Services to invoke. In this project the Web Service is email, however the code provides extendibility to use different strategies.
2. The ***WebServiceInvocationWrapper*** and ***WebServiceQuery*** classes utilize the Singleton design pattern to ensure only one instance of these objects are present at an instance in time.
3. Many calls to the JainSIP libraries utilize the Factory design pattern in where a concrete product that implements the product interface is returned. (***SipFactory, MessageFactory, HeaderFactory***, etc)
4. The Web Service query employed the design of Iterator to effectively to collect information regarding ***serviceInfoColl***’s specific service objects.

# Architecture

The software architecture pattern employed in this system is the **Client Server architecture**. The Client and Server (Sip Communicator and Proxy) communicate via communication subsystems through Reply/Request messages and listeners that process them.

Below is the package level component diagram of the system [Figure 1]. Please not that due to space constraints only a highlight of interface methods is shown. The rest of the methods can be found in Tables 1 and 2.

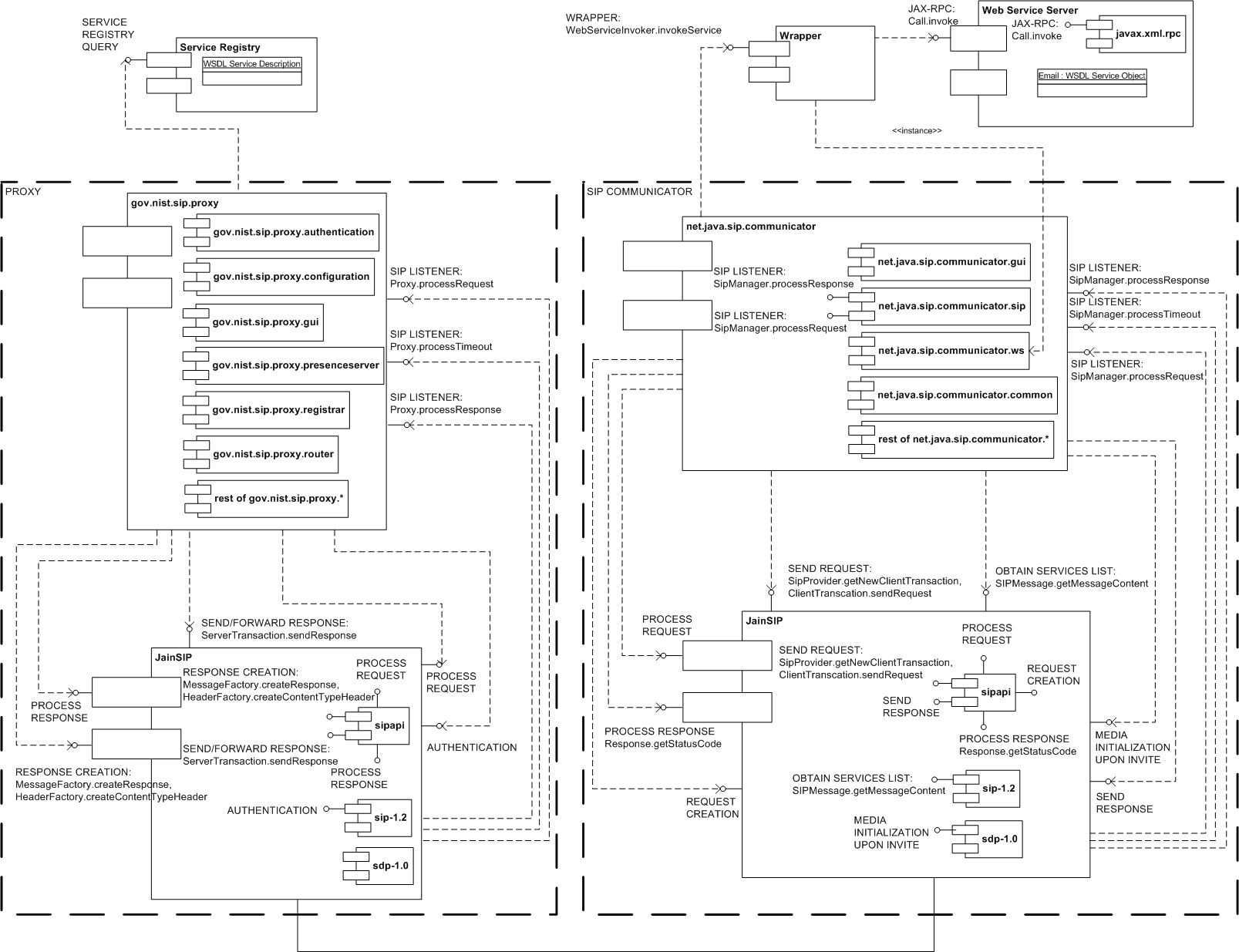


Figure 1: Component Diagram

Table 1: Interface Methods for Proxy (Server) and Service Registry

|  |  |  |
| --- | --- | --- |
| **Service Registry** | SERVICE REGISTRY QUERY | WebServiceQuery.getInstance |
|  |  | WebServiceQuery.findServicesForOrg |
|  |  | ServiceInfoColl.size |
|  |  | ServiceInfoColl.getDescription |
|  |  | ServiceInfoColl.getWsdluri |
|  |  | ServiceInfoColl.getEndPoint |
|  |  |  |
| **gov.nist.sip.proxy** | SIP LISTENER | Proxy.processRequest |
|  |  | Proxy.processResponse |
|  |  | Proxy.processTimeOut |
|  |  |  |
| **JainSip (Proxy)** | AUTHENTICATION | AuthorizationHeader.getNonce |
|  |  | AuthorizationHeader.getRequestURI |
|  |  | AuthorizationHeader.getResponse |
|  |  | AuthorizationHeader.getRealm |
|  |  | AuthorizationHeader.getUserName |
|  | PROCESS RESPONSE | ResponseEvent.getResponse |
|  |  | ResponseEvent.getClientTransaction |
|  |  | Response.getStatusCode |
|  |  | Response.getReasonPhrase |
|  | PROCESS REQUEST | RequestEvent.getRequest |
|  |  | RequestEvent.getServerTransaction |
|  |  | Request.getMethod |
|  |  | Request.getRequestURI |
|  | RESPONSE CREATION | MessageFactory.createResponse |
|  |  | HeaderFactory.createContentTypeHeader |
|  | SEND/FORWARD RESPONSE | ServerTransaction.sendResponse |
|  |  | SipProvider.sendResponse |

Please note that these methods are still a subset of the total methods that allows the interaction between JainSIP and the proxy. These methods are methods that are of some significance and are on the main path of code assuming the Web Services query will be successful and the caller issues an invite to the callee who have registered Web Services.Table 2: Interface Methods for Sip Communicator (Client), Wrapper, and Web Services Server

|  |  |  |
| --- | --- | --- |
| **Wrapper** | WRAPPER | WebServiceInvoker.invokeService |
|  |  |  |
| **Web Services Server** | INVOKE WEB SERVICE | Call.invoke |
|  |  |  |
| **net.java.sip.communicator** | SIP LISTENER | Proxy.processRequest |
|  |  | Proxy.processResponse |
|  |  | Proxy.processTimeOut |
|  |  |  |
| **JainSip (Proxy)** | MEDIA INITIALIZATION UPON INVITE | SdpFactory.createSessionDescription |
|  |  | SdpFactory.createOrigin |
|  |  | SdpFactory.createSessionName |
|  |  | SdpFactory.createConnection |
|  |  | SdpFactory.createTimeDescription |
|  |  | SdpFactory.createMediaDescription |
|  | PROCESS REQUEST | RequestEvent.getRequest |
|  |  | RequestEvent.getServerTransaction |
|  |  | Request.getMethod |
|  |  | Message.getHeader |
|  |  | CSeqHeader.getMethod |
|  |  | Transaction.getDialog |
|  | PROCESS RESPONSE | ResponseEvent.getResponse |
|  |  | ResponseEvent.getClientTransaction |
|  |  | Response.getStatusCode |
|  |  | Response.getReasonPhrase |
|  |  | Message.getHeader |
|  |  | CSeqHeader.getMethod |
|  |  | Dialog.sendAck |
|  |  | Dialog.sendRequest |
|  | REQUEST CREATION | MessageFactory.createRequest |
|  |  | HeaderFactory.createCSeqHeader |
|  |  | HeaderFactory.createFromHeader |
|  |  | HeaderFactory.createViaHeader |
|  |  | HeaderFactory.createMaxForwardsHeader |
|  |  | HeaderFactory.createContactHeader |
|  |  | HeaderFactory.createToHeader |
|  |  | HeaderFactory.createContentTypeHeader |
|  |  | AddressFactory.createURI |
|  |  | AddressFactory.createAddress |
|  |  | AddressFactory.createSipURI |
|  |  | SipProvider.getNewCallId |
|  |  | Message.setContent |
|  | SEND REQUEST | SipProvider.getNewClientTransaction |
|  |  | ClientTranscation.sendRequest |
|  | SEND RESPONSE | SipProvider.getNewServerTransaction |
|  |  | ServerTransaction.sendResponse |
|  | OBTAIN SERVICES LIST | SIPMessage.getMessageContent |

Please note that these methods are still a subset of the total methods that allows the interaction between JainSIP and the Sip Communicator assuming the Web Services query will be successful and the caller issues an invite to the callee who have registered Web Services.

The following figure [Figure 2] illustrates the deployment diagram of the system with the current run-time configurations involving the client and proxy, along with the application server all on one node.

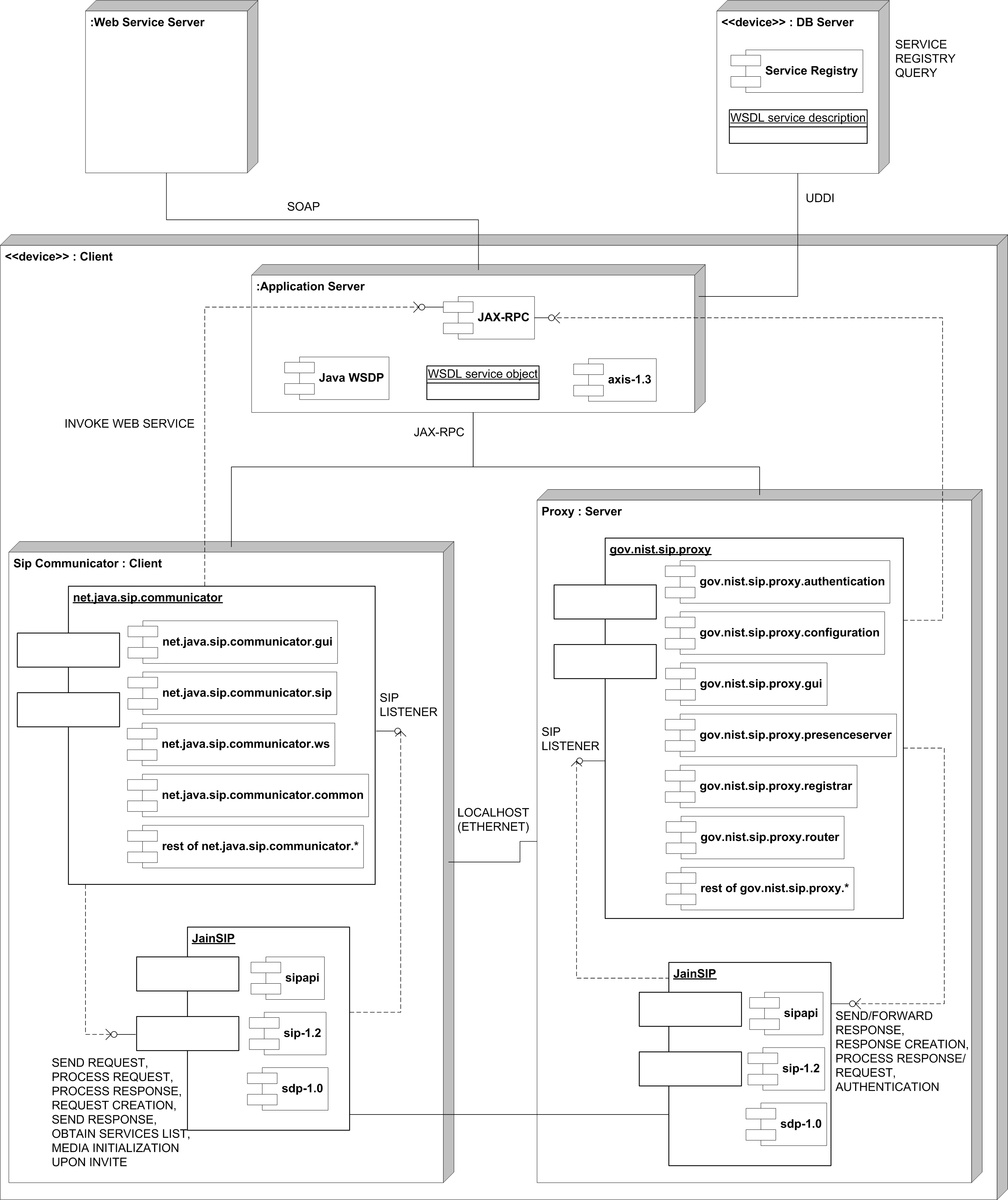


Figure 2: Deployment Diagram

# Detailed Class Diagrams

## UML Class Diagrams

The following is the UML class diagram for the Sip Communicator that pertains to the modification in the class ***WSEvent*** [Figure 3]. The Wrapper classes are also included in this diagram as they are also part of the Sip Communicator.

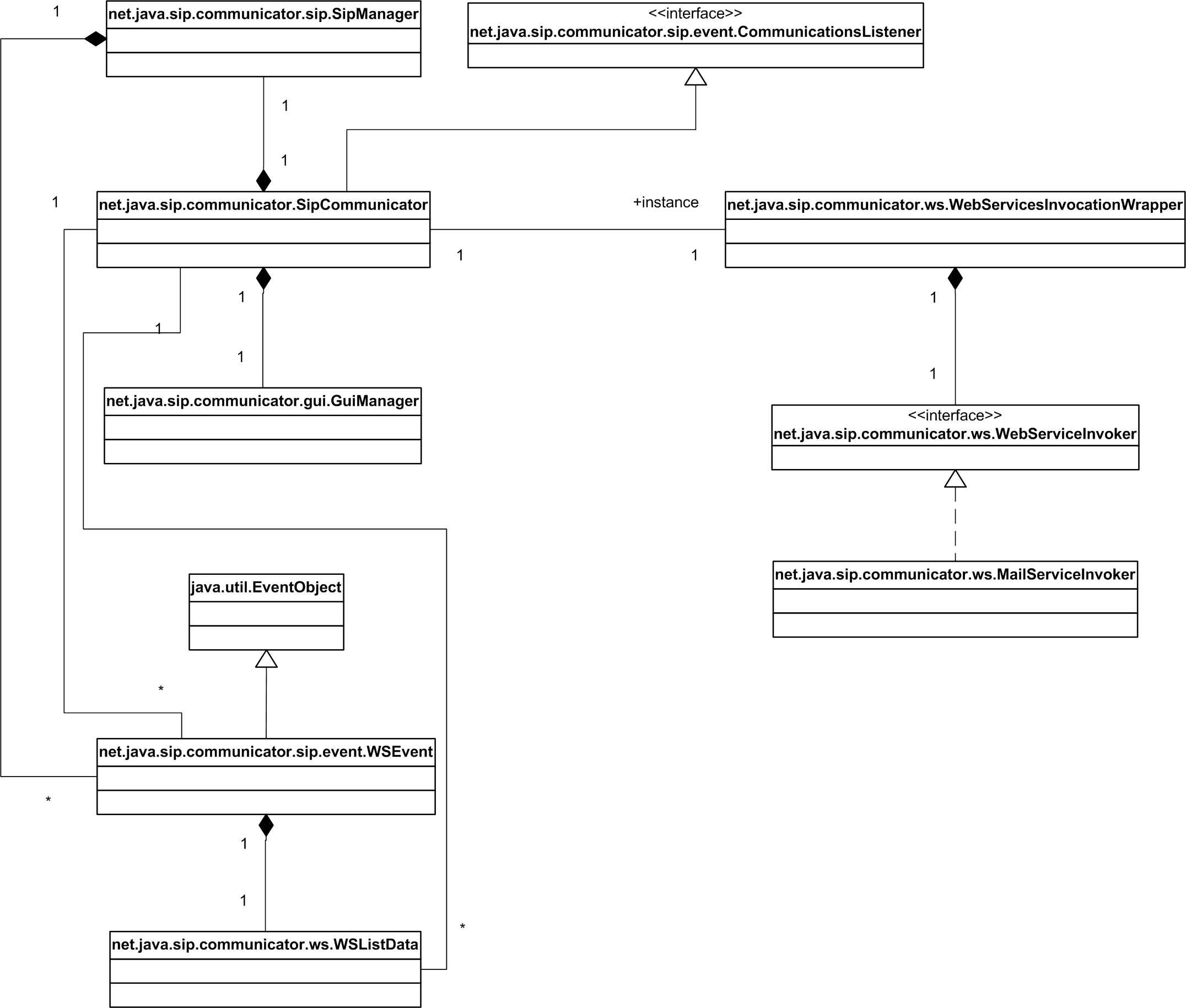


Figure 3: UML Class Diagram for Sip Communicator and Wrapper

Below is a figure showing the attributes and methods of the modified class ***WSEvent*** [Figure 4].

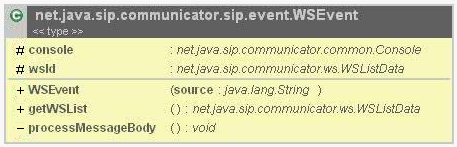


Figure 4: ***WSEvent*** class details

Several design patterns are evident in the Sip Communicator.

The following diagram [Figure 5] shows a **Strategy Software Design Pattern** that is evident in the classes of the wrapper. A family of algorithms in the form of the specific type of ***WebServiceInvoker*** is encapsulated and can be used by the client interchangeably. For example, various implementations for the ***WebServiceInvoker*** can exist, in our case only the email strategy is used but this can be easily extended by adding different strategies.

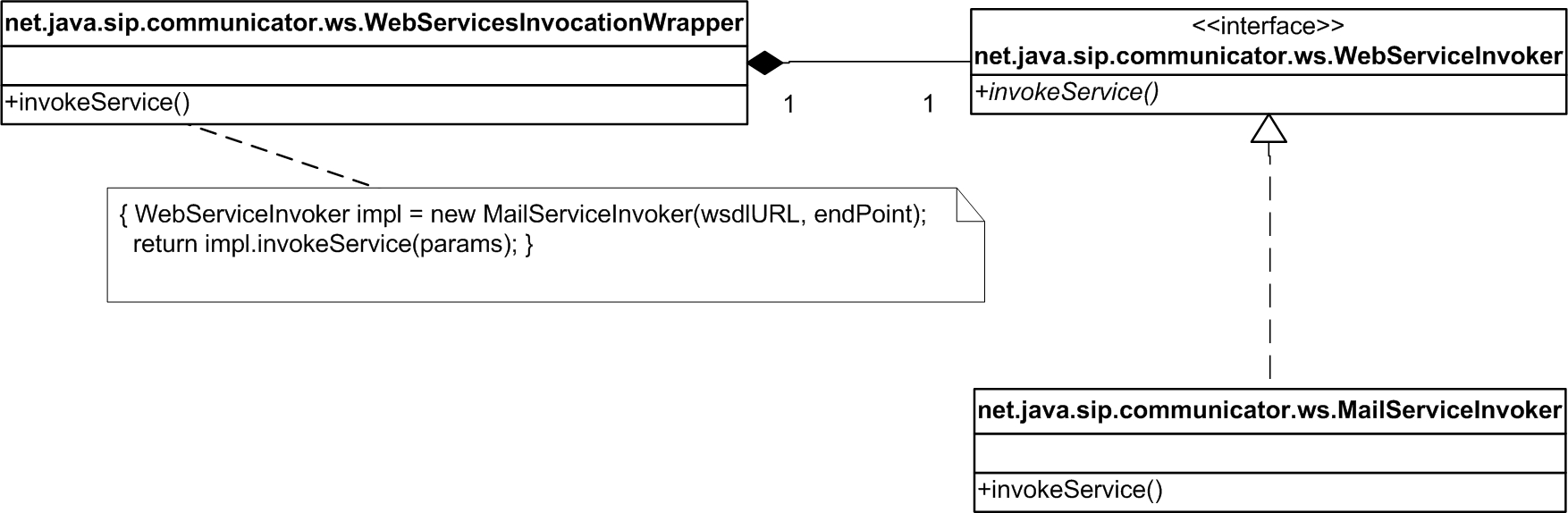


Figure 5: Strategy Design Pattern in Sip Communicator

The following diagram [Figure 6] shows a **Singleton Software Design Pattern** that is evident in the class ***WebServicesInvocationWrapper***. The code insures that this class only has one instance and provide a global access point to it through the method ***getInstance()***. This is important as only one instance of the wrapper for a client can exist.

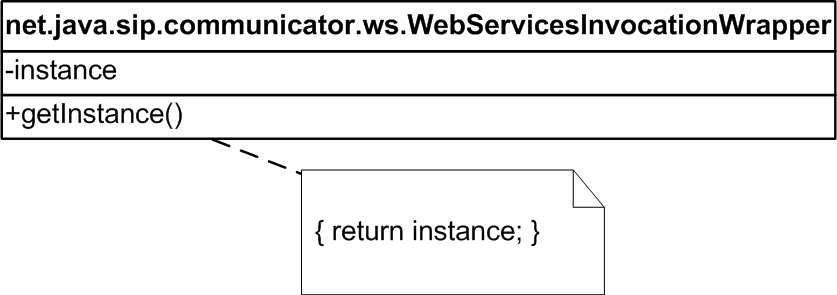


Figure 6: Singleton Design Pattern in Sip Communicator

The following diagram [Figure 7] shows two examples of the **Factory Software Design Pattern** that is evident in the Sip Communicator. This design pattern define an interface for clients to use to create an object, but lets subclasses decide which class to instantiate. Factory Methods let a class defer instantiation to subclasses. For example, in the case of the ***ServiceFactory*** in the ***MailServiceInvoker*** and the ***SipFactory*** in the ***SipManager***, as long as a concrete product is received that implements the product; the client does not care how the class is instantiated.

There are many instances where this pattern is used, only two examples are shown.

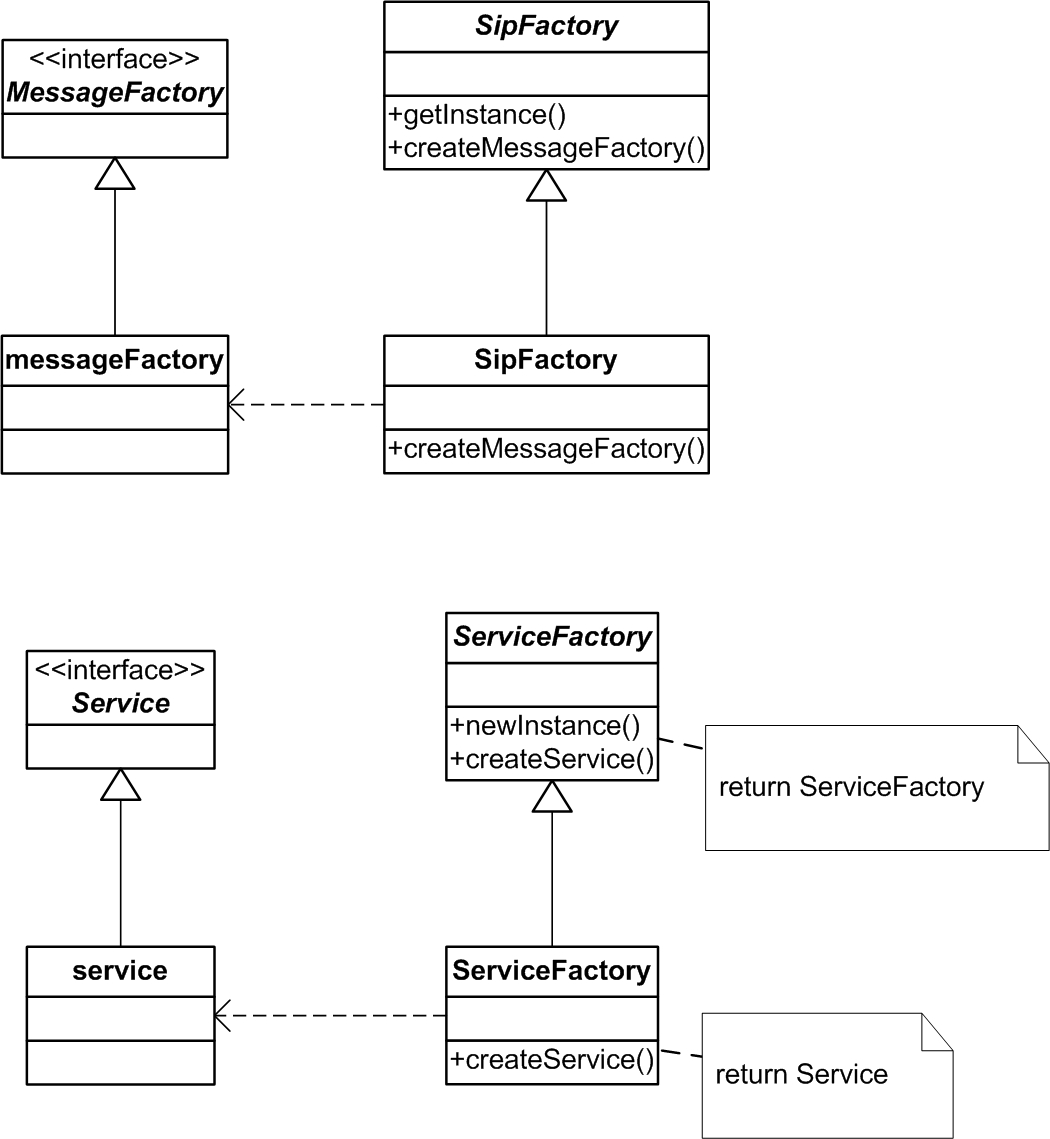


Figure 7: Factory Design Pattern in Sip Communicator

The following is the UML class diagram for the Sip Proxy that pertains to the modification in the class ***Proxy*** [Figure 8].

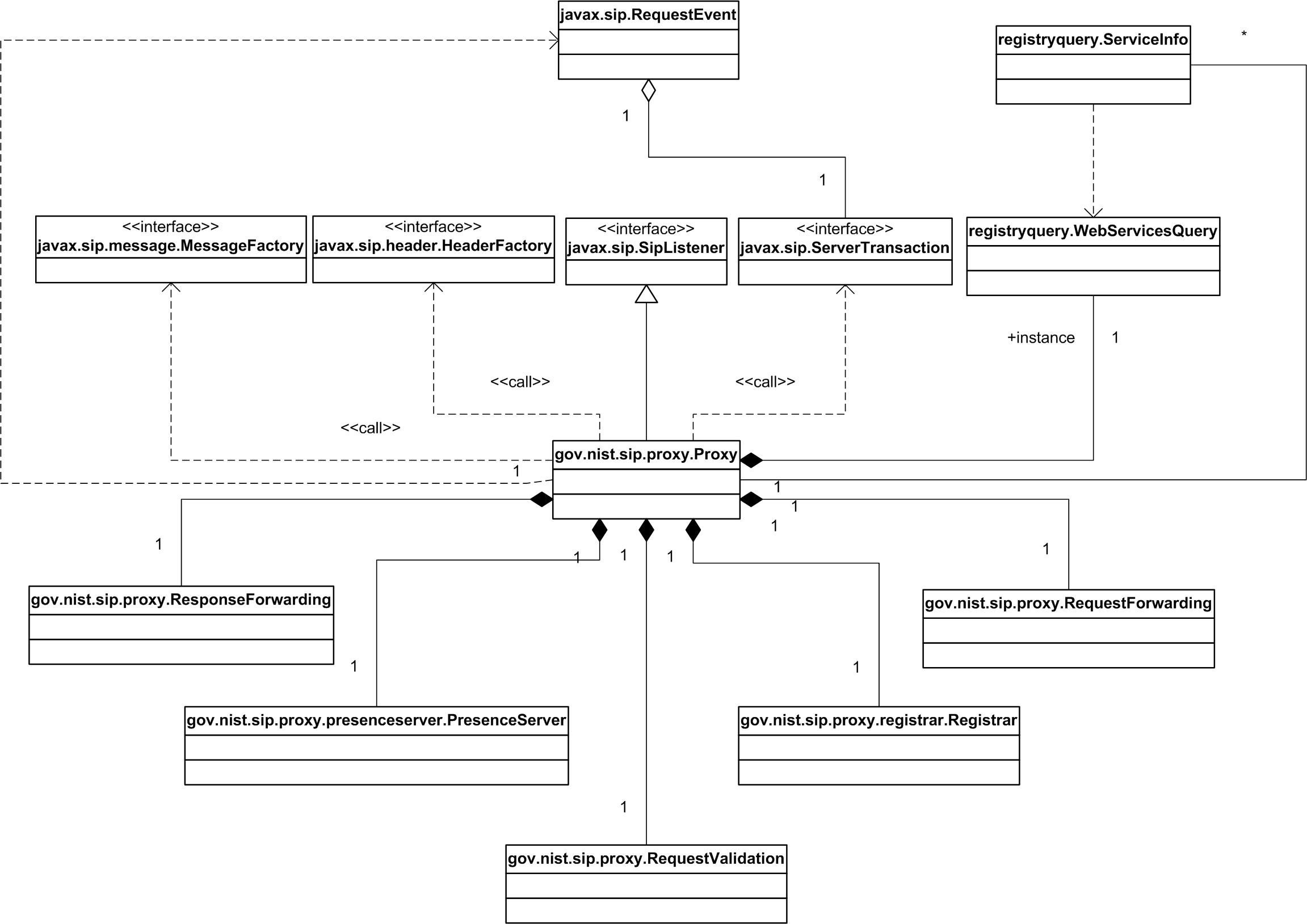


Figure 8: UML Class Diagram for Sip Proxy

Below is a figure showing the attributes and methods of the modified class ***Proxy*** [Figure 9].



Figure 9: ***Proxy*** class details

Several design patterns are also evident in the Sip Proxy.

The following diagram [Figure 10] shows a **Singleton Software Design Pattern** that is evident in the class ***WebServicesQuery***. The code insures that this class only has one instance and provide a global access point to it through the method ***getInstance()***. This is important as only one instance of the Web Services query exist for a proxy since the query is to an external database.

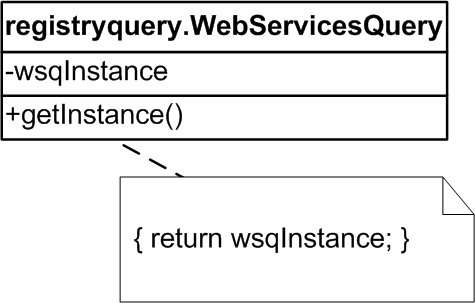


Figure 10: Singleton Design Pattern in Sip Communicator

The following diagram [Figure 11] shows an **Iterator Software Design Pattern** that has been implemented in the class ***Proxy*** for the processing of the Web Service query result. This design pattern allows the access of elements of an aggregate (***serviceInfoColl*** representing objects containing the service info description, URI, endpoints, etc) sequentially without exposing its representation. The Proxy uses the integrator design pattern to collect information regarding ***serviceInfoColl***’s specific service objects.

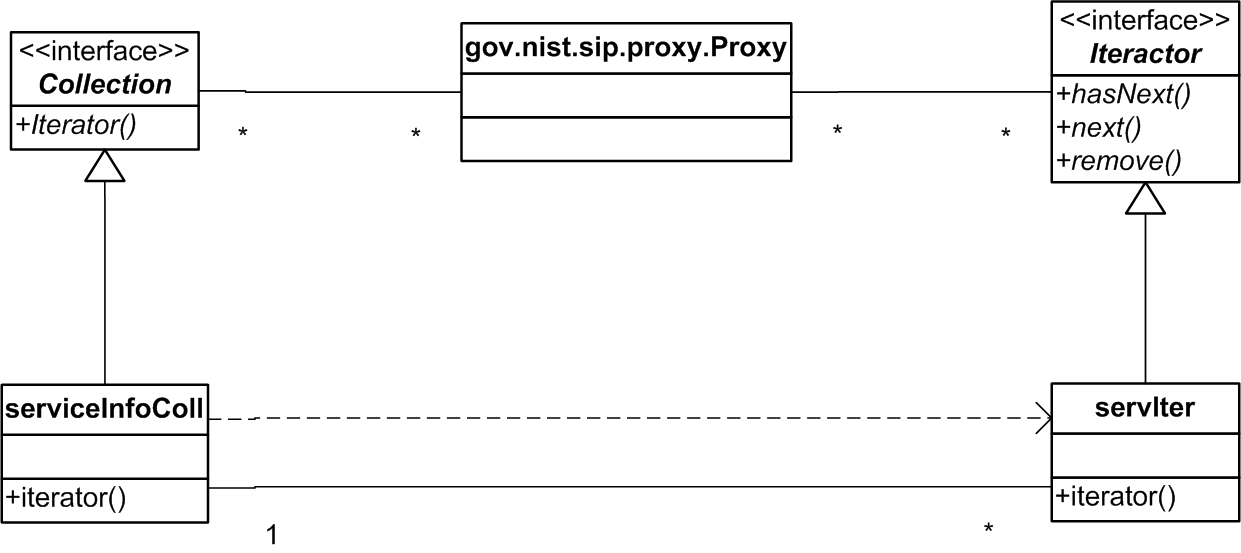


Figure 11: Iterator Design Pattern in Sip Communicator

## Method Details

The code to ***Proxy*** has been modified to be able to query the Service Registry and pass a list of Web Services or NULL to the proxy for processing. This occurs at the point where the proxy requests the called agent’s location. If no services are found for the agent then the proxy operation proceeds normally forwarding the message to the Callee. However if services are found, the proxy builds a DECLINE response with message content of the type *<Number-of-services:Integer> -- <Service1-description:String> <Service1-Wsdluri::URL> <Service1-EndPoint> -- <Service2-description:String> <Service2-Wsdluri::URL> <Service2-EndPoint>* and sends this response back to the Caller. Below is the pseudo-code for this modification in Proxy.

public void processRequest(RequestEvent requestEvent)

...

if registrar has registration for request

...

if targetURIList is not null and targetURIList is not empty

// Parse the organization name from URI

String orgName = String.split() the request.getRequestURI() to obtain orgName

// Get services info for organization using WebServicesQuery

WebServicesQuery wsq = get Instance from WebServicesQuery Collection serviceInfoColl = wsq.findServicesForOrg(orgName);

if serviceInfoColl is not null and serviceInfoColl’s size is not 0

wsdl\_list = serviceInfoColl's size, i.e number of services

Iterator servIter = serviceInfoColl.iterator();

while servIter has another element

ServiceInfo servInfo = servIter's next element

wsdl\_list add servInfo’s Description, Wsdluri, Endpoint

// Create DECLINE message with service list

ContentTypeHeader contentTypeHeader = headerFactory

.createContentTypeHeader("text", "plain"); // Create response

Response response = messageFactory.createResponse(Response.DECLINE, request, contentTypeHeader, wsdl\_list);

// Send response

if serverTransaction is not null

serverTransaction.sendResponse(response);

else sipProvider.sendResponse(response);

else

Forward the request statefully to each target

The code to ***WSEvent*** has been modified to be able to accept the message content from the proxy is a DECLINE message is received of the form *<Number-of-services:Integer> -- <Service1-description:String> <Service1-Wsdluri::URL> <Service1-EndPoint> -- <Service2-description:String> <Service2-Wsdluri::URL> <Service2-EndPoint>* and process this formatted message to create a ***WSListData*** object and reference it by wsdl private member to be used by the Sip Communicator and GUI. Below is the pseudo code.

private void processMessageBody()

...

try

//parse <Number-of-services > -- <Service1-description > <Service1-Wsdluri > <Service1- //EndPoint> -- <Service2-description > <Service2-Wsdluri > <Service2-EndPoint> into //WSListData object assuming correct format

if messageContent is null or messageContent's length == 0

return;

String [] pattern = split messageContent into 2 elements corresponding to first " -- "

// \*\*now <Number-of-services>, <Service1-description > <Service1-/Wsdluri > <Service1- //EndPoint> -- <Service2-description> <Service2-Wsdluri> <Service2-EndPoint>

String numberServicesString = pattern[0], i.e. <Number-of-services>

numberServices = convert numberServicesStringto integer

String servicesList = pattern[1], i.e. the actual list

String [] serviceListVector = split the list into separate services corresponding to " -- " [servicesList.split("--", numberServices)]

// \*\*now <Service1-description> <Service1-Wsdluri> <Service1-EndPoint>, <Service2- //description> <Service2-Wsdluri> <Service2-EndPoint>

for i = 0; i < numberServices; i++

// Get the corresponding service description, wsdl, endpoints

String serviceList = serviceListVector[i]

String [] serviceObject = split corresponding to spaces

// \*\*now <Service1-description>, <Service1-Wsdluri>, <Service1-EndPoint>

wsdl = second last element of serviceObject

endPoint = last element of serviceObject

desc = remaining serviceObject

// Create the wsdl object and add the service

wsdl = new WSListData

wsdl.addWSItem(wsdl, desc, endPoint)

catch (Exception ex)

print error

# State Diagrams

Below is the state diagram for the ***Call*** class [Figure 12]. Please note that the format of the transition is Event [Condition] / Action and the listed action is usually a private method call to change the state. Also note that italicized Actions and States are usually caused by the proxy server’s response while non-italicized Actions are usually caused by the Calling/Callee Agent’s GUI interactions.

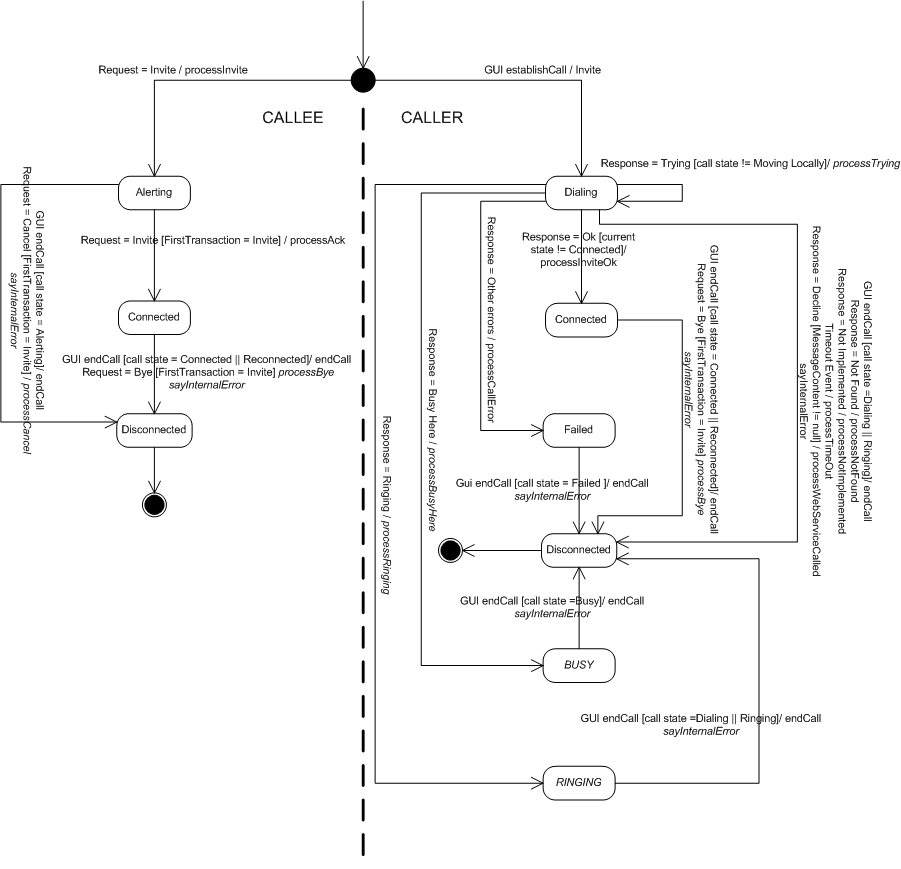


Figure 12: State Diagram for ***Call*** class

# Open Issues

There are currently no open issues pertaining to the design of this software on the current configuration. Implementation and testing may possibly yield additional considerations that might affect the proposed design and pseudo code.

# Domain Dictionary

## Terms and Abbreviations

|  |  |
| --- | --- |
| Term | Definition |
| SIP | Session Initiation Protocol, is a signaling protocol for Internet conferencing, telephony, presence, events notification and instant messaging |
| JAIN-SIP | Java API for Integrated Networks Session Initiation protocol |
| WSDL | WSDL (Web Services Description Language) is an XML-based language for describing Web services and how to access them. |