IMPLEMENTACIÓN DE JTAPI, PARA LA PLATAFORMA DHARMA DE DATAVOICE.

1. Introducción a JTAPI.
   - 1.1. ¿ Qué es JTAPI ?.
   - 1.2. ¿ Qué puedo hacer con JTAPI ?.
   - 1.3. Ventajas de utilizar JTAPI.
   - 1.4. Inconvenientes de utilizar JTAPI.
   - 1.5. ¿ Qué versión del estándar JTAPI es implementada por DataVoice ?
   - 1.6. ¿ Donde puedo obtener extensa documentación del estándar JTAPI ?
2. Estructuración de JTAPI
   - 2.1. Paquetes que lo componen.
     - 2.1.1. Paquete Core.
       - 2.1.1.1. Introducción.
       - 2.1.1.2. Ejemplos
         - 2.1.1.2.1. Realizar una llamada.
         - 2.1.1.2.2. Contestar una llamada.
       - 2.1.1.3. Nivel de implementación del estándar por parte de DataVoice.
       - 2.1.1.4. Diferencias y comentarios de la implementación de DataVoice con respecto al estándar.
     - 2.1.2. Paquete CallControl
       - 2.1.2.1. Introducción
       - 2.1.2.2. Ejemplos.
         - 2.1.2.2.1. Transferir una llamada
         - 2.1.2.2.2. Conferenciar una llamada.
         - 2.1.2.2.3. Descolgar una llamada en un terminal diferente del que está sonando el timbre.
       - 2.1.2.3. Nivel de implementación del estándar por parte de DataVoice.
       - 2.1.2.4. Diferencias y comentarios de la implementación de DataVoice con respecto al estándar.
     - 2.1.3. Paquete CallCenter
       - 2.1.3.1. Introducción
       - 2.1.3.2. Ejemplos.
         - 2.1.3.2.1. Número de agentes de todos los grupos de agentes.
         - 2.1.3.2.2. Dar de alta a un agente en un grupo
         - 2.1.3.2.3. Asociar datos a una llamada
       - 2.1.3.3. Nivel de implementación del estándar por parte de DataVoice.
       - 2.1.3.4. Diferencias y comentarios de la implementación de DataVoice con respecto al estándar.
     - 2.1.4. Paquete Phone.
       - 2.1.4.1. Introducción.
       - 2.1.4.2. Ejemplos.
         - 2.1.4.2.1. Observar el display de un puesto.
         - 2.1.4.2.2. Subir el volumen del altavoz de un puesto.
         - 2.1.4.2.3. Presionar las teclas de un puesto
       - 2.1.4.3. Nivel de implementación del estándar por parte de DataVoice.
       - 2.1.4.4. Diferencias y comentarios de la implementación de DataVoice con respecto al estándar.
   - 2.2. Instalación de los paquetes necesarios para desarrollar aplicaciones basadas en JTAPI.
   - 2.3. Fichero de configuración para las aplicaciones clientes.
3. Instalación del servidor JTAPI.
   - 3.1. Introducción.
   - 3.2. Requerimientos necesarios para instalar el servidor.
   - 3.3. ¿ Como ejecuto el servidor ?
   - 3.4. ¿ Cómo paro el servidor ?
   - 3.5. ¿ Puedo arrancar varios servidores ?
4. Consejos de programación con JTAPI.

1. Introducción a JTAPI

1.1 ¿ Qué es JTAPI ?

JTAPI (Java Telephony Application Programmming Interface), es un interface de programación para desarrollar utilizando el lenguaje de programacion Java, aplicaciones de telefonía.

1.2 ¿ Qué puedo hacer con JTAPI ?

Con JTAPI puedo realizar todo tipo de aplicaciones CTI (Computer Telephony Integration), es decir aplicaciones que interactuen con una plataforma de telefonía para por ejemplo, realizar llamadas, transferir llamadas, conferenciar llamadas, controlar a los agentes de los grupos de agentes, realizar llamadas predictivas, etc.

1.3 Ventajas de utilizar JTAPI.

Las aplicaciones JTAPI son portables entre plataformas de telefonía, es decir, una aplicación basada en JTAPI, funcionará en distintas plataformas de telefonía, sin tener si quiera que recompilar el código fuente. Además, las aplicaciones podrán ejecutarse en una gran variedad de entornos informáticos, es decir, funcionarán sobre diferentes sistemas operativos y arquitecturas. ¡ Sin tener que recompilar el código !.

¡ Imagine el ahorro de costes que supone para una empresa !. Una aplicación se escribirá una vez y podrá ser vendida a los clientes, independientemente de su entorno de ejecución.

JTAPI, fue pensado por un grupo de expertos en CTI, dando como resultado un API potente y sencillo de aprender, por lo que en poco tiempo será capaz de realizar complejas aplicaciones de telefonía.

Otra gran ventaja que aporta JTAPI, es un que dispondrá de toda la información en tiempo real. Por ejemplo, podrá saber cuantas llamadas hay en un determinado momento, cuantos agentes hay en un determinado grupo, que pone en el display de un puesto, cuantos agentes están trabajando en un determinado momento, etc.

1.4 Inconvenientes de utilizar JTAPI.

Las aplicaciones basadas en JTAPI tiene el inconveniente de que al ser escritas en Java, pueden ser algo más lentas, al ser Java un lenguaje de programación interpretado (no compilado). Pero con las potentes máquinas actuales, no será un inconveniente suficiente para descartarlo.

1.5 ¿ Qué versión del estándar es implementado por DataVoice ?

Cómo la mayoría de los APIs y componentes en el mundo informático. Estos se van mejorando y/o completando en distintas versiones. DataVoice implementa la versión 1.3 de JTAPI, que es una completa y potente versión del estándar.

1.6 ¿ Donde puedo obtener extensa documentación del estándar JTAPI ?

A continuación, en este documento, se realizarán comentarios y observaciones de JTAPI, pero podrá disponer de una extensa documentación en distintos formatos en la siguiente dirección http://java.sun.com/products/jtapi/index.html.

2. Estructuración de JTAPI

Cómo la mayoría de los APIs de programación, estos están estructurados en diferentes paquetes. Aportando cada uno de ellos servicios específicos.

2.1 Paquetes que lo componen.

JTAPI está dividido en diferentes paquetes de los cuales DataVoice implementa los paquetes Core, CallCenter, CallControl y Phone. (Los más importantes, y suficientes para la gran mayoría de aplicaciones).

2.1.1 Paquete javax.telephony.Core.

2.1.1.1 Introducción

Con este paquete podremos:

A) Realizar, finalizar y contestar llamadas.
B) Obtener el estado en el que se encuentran el proveedor, las llamadas, los componentes de la llamada.
C) Obtener los terminales y direcciones que forman parte del dominio del entorno telefónico.
D) Poner oyentes a las llamadas, terminales, proveedor, etc. para que se nos avise mediante la invocación de un método que ha sucedido algo. Por ejemplo la llamada a finalizado, o el proveedor ha dejado de estar en servicio.

2.1.1.2 Ejemplos

2.1.1.2.1 Realizar una llamada.

```
import javax.telephony.*;

public class RealizarLlamadaApp {

        public static void main(String[] args){
                Provider            prov;
                Terminal            term;
                Call         call;
                JtapiPeer           peer;
                Connection[]  connections;
                CallObserver callObsv;

                // Verificamos los argumentos. Deben ser dos números de teléfono.
                if ( args.length != 2 ){
                     System.out.println("Formato:\"RealizarLlamadaApp <origen>
                                                                    <destino>\"");
                     System.exit(1);
                }

                try {
                    // Obtenemos una instancia que implemente javax.telephony.JtapiPeer.
                   peer  = JtapiPeerFactory.getJtapiPeer(null);

                   // Obtenemos un proveedor que de el servicio. Por defecto Core.
                   prov  = peer.getProvider(null);


                  // Instanciamos y configuramos los objetos necesarios para la llamada
                  term  = prov.getTerminal(args[0]);
                  call    = prov.createCall();

                  // Realizamos una llamada
                 connections = call.connect(term, term.getAddresses()[0], args[1]);

                  // Finalizamos el proveedor
                 prov.shutdown();

        } catch ( Exception e ){
                System.out.println(e.toString());
        }

        // Terminamos la aplicación
        System.exit(0);
        }
}
```

2.1.1.2.2 Contestar una llamada

```
////////////////////////////////////////////////////////////////////////////////
//// Descuelga todas las llamadas que suenen en el puesto que se da como argumento

import javax.telephony.*;
import javax.telephony.TerminalConnection;
import javax.telephony.events.CallEv;
import javax.telephony.events.TermConnRingingEv;

public class AnswerCallApp {

	public static void main(String args[]){
		JtapiPeer	peer		= null;
		Provider	prov		= null;
		Terminal	term		= null;
		java.lang.String device	= null;
		CallObserver	myCallObserver = null;

		try {

			if ( args.length != 1 ){
			    System.out.println("Formato \"AnswerCallApp <puesto>\"" );
			    System.exit(1);
			}

			// Obtenemos la implementación javax.telephony.JtapiPeer.
			peer  = JtapiPeerFactory.getJtapiPeer(null);

			// Obtenemos un proveedor que de el servicio. Por defecto Core.
			prov  = peer.getProvider(null);

			// Obtenemos el terminal del cual vamos a descolgar las llamadas.
			term = prov.getTerminal(args[0]);

			myCallObserver = new MyCallObserver();
			term.addCallObserver(myCallObserver);

			Thread.sleep(25000);  // Descuelga las llamadas durante un tiempo

			term.removeCallObserver(myCallObserver);

		} catch ( Exception e ){
			System.out.println( e.toString() );
		}

		// Finalizamos el proveedor.
		try {
			prov.shutdown();
		} catch (Exception e){}

		// Salimos
		System.exit(0);
	}
}

class MyCallObserver implements CallObserver {

	// Implementación del método de la interfaz CallObserver
	public void callChangedEvent(CallEv[] ev){
		TerminalConnection tc;

		for ( int i = 0; i < ev.length; i++ ){
			if ( ev[i].getID()  != TermConnRingingEv.ID )
				continue;
			tc = ((TermConnRingingEv) ev[i]).getTerminalConnection();
			try {
				tc.answer();
			} catch ( Exception e ){
				// Nunca se datá
			}
		}
	}
}
```

2.1.1.3 Nivel de implementación del estándar por parte de DataVoice.

DataVoice implementa completamente este paquete.

2.1.1.4 Diferencias y comentarios de la implementación de DataVoice con respecto al estándar.

No hay diferencias entre la implementación realizada por JTAPI y el estándar. Lo único que debe saber el programador es que el mapa de objetos se va completando dinámicamente a medida que va transcurriendo el tiempo.

Es decir, por ejemplo. Los puestos que forman parte del dominio del proveedor inicialmente cuando el servidor es arrancado estará vacío por lo que el método
public Terminal[] getTerminals() de la clase Provider devolverá null, pero a medida que pase el tiempo se irá completando.

Un terminal sólo tiene una dirección y viceversa.

2.1.2 Paquete javax.telephony.CallControl

2.1.2.1 Introducción

Con este paquete podremos, además de todo lo especificado en el paquete Core:

a) Transferir llamadas (Directas e indirectas)
b) Conferenciar llamadas
c) Redireccionar llamadas.
d) Descolgar llamadas que están sonando en otro terminal.
e) Realizar llamadas de consulta.
f) Añadir un puesto para que forme parte de una conferencia ya establecida.
g) Retener llamadas
h) Recuperar llamadas retenidas.

Además dispondrá.

A) Estados más finos sobre para las llamadas y sus componentes.
B) Información sobre el número llamante y llamado de una llamada.

2.1.2.2 Ejemplos

2.1.2.2.1 Transferir una llamada

```
////////////////////////////////////////////////////////////////////////////////
/// Ejemplo sobre transferencias. Pasos a seguir:
///            - Realizamos una llamada del "662" al "228887766" y esperamos un tiempo
///              ( Suponemos que descuelga )
///            - Retenemos la llamada.
///            - Realizamos una llamada del "662" al "648"
///            - Realizamos la transferencia.

import javax.telephony.*;
import javax.telephony.events.*;
import javax.telephony.callcontrol.*;
import javax.telephony.callcontrol.events.*;

public class TransferApp {

          public static void main(String[] args){
               JtapiPeer             peer  = null;
               Provider              prov  = null;
               CallControlCall       call   = null;
               CallControlCall       call2 = null;
               Terminal              term = null;
               TerminalConnection[]  tcs    = null;
               Connection[]          cons = null;


               try {
                  peer = JtapiPeerFactory.getJtapiPeer(null);
                  prov = peer.getProvider(null);

                  term = prov.getTerminal("662");
                  call = (CallControlCall) prov.createCall();
                  call.connect(term, term.getAddresses()[0], "228887766");

                  // Esperamos unos segundos en los que debe descolgar.
                  Thread.sleep(4000);

                  // Suponemos que sólo hay una llamada activa en el terminal.
                  call2 = (CallControlCall) prov.createCall();
                  tcs  = call.getConnections()[0].getTerminalConnections();
                  cons = call2.consult(tcs[0], "648");

                  call.setTransferController(call.getConnections()[0].
                                             getTerminalConnections()[0]);
                  call2.setTransferController(call2.getConnections()[0].
                                             getTerminalConnections()[0]);
                  call.transfer(cons[0].getCall());

                  Thread.sleep(1000);
                  prov.shutdown();
               } catch ( Exception e ){
                  System.out.println(e.toString());
               }
               System.exit(0);
       }
}
```

2.1.2.2.2 Conferenciar una llamada

```
////////////////////////////////////////////////////////////////////////////
/// Este ejemplo realiza una llamada del "662" al "228887766". Luego
/// realiza una llamada del "662" al "648" y pone en conferencia las dos llamadas.
/// Muestra por pantalla los eventos que se reciben de las llamadas implicadas

import javax.telephony.*;
import javax.telephony.events.*;
import javax.telephony.callcontrol.*;
import javax.telephony.callcontrol.events.*;

public class ConferenceCallsApp {

        ////////////////////////////////////////////////////////////////////////////

        public static void main(String args[]){
             JtapiPeer         peer;
             Provider          prov;
             CallControlCall   call1, call2;
             Connection[]      connections1;
             Connection[]      connections2;
             TerminalConnection[] tcs;


             try {
              peer  = JtapiPeerFactory.getJtapiPeer(null);
              prov  = peer.getProvider(null);

              call1 = (CallControlCall) prov.createCall();
              connections1 = call1.connect( prov.getTerminal("662"),
                                            prov.getAddress("662"), "648");

              Thread.sleep(1000);

              System.out.println("Realizando la segunda llamada");
              call2 = (CallControlCall) prov.createCall();
              connections2 = call2.connect(prov.getTerminal("662"),
                                        prov.getAddress("662"),
                                        "228887766");

              Thread.sleep(1000);

              // Configuramos los controladores de conferencias
              call1.setConferenceController(connections1[0].getTerminalConnections()[0]);

              call2.setConferenceController(connections2[0].getTerminalConnections()[0]);

              ((CallControlTerminalConnection)call1.getConferenceController()).unhold();

              call1.conference(call2);

              Thread.sleep(1000);

              call1.drop();
              call2.drop();

              prov.shutdown();
             } catch ( Exception e ){
                 System.out.println(e.toString());
             }

             System.exit(0);
        }
}
```

2.1.2.2.3 Descolgar una llamada en un terminal diferente del que está sonando el timbre

```
////////////////////////////////////////////////////////////////////////////////
/// Esta aplicación muestra el uso de CallControlTerminal.pickup().
///Todas las llamadas que suenen en 662 serán pasadas al 648 durante 2 minuto.
/// Las llamadas al 662 las hacemos a mano y deben ser llamadas que vengan de la calle.

import javax.telephony.*;
import javax.telephony.events.*;
import javax.telephony.callcontrol.*;
import javax.telephony.callcontrol.events.*;


public class PickUpApp {

        public static void main(String[] args){
                JtapiPeer       peer;
                Provider        prov;
                Address         addr;
                CallObserver    callObserver;

                try {
                   peer = JtapiPeerFactory.getJtapiPeer(null);
                   prov = peer.getProvider(null);

                   callObserver = new MyCallObserver((CallControlTerminal)
                                        prov.getTerminal("648"));
                   addr         = prov.getAddress("662");
                   addr.addCallObserver(callObserver);

                   Thread.sleep(2 * 1000 * 60);

                   prov.shutdown();

                   System.exit(0);
                } catch ( Exception e ){
                   System.out.println(e.toString());
                }
        }
}

class MyCallObserver implements CallObserver {

        private CallControlTerminal term;

        public MyCallObserver(CallControlTerminal term)  {
                this.term = term;
        }

        public void callChangedEvent(CallEv[] evlist) {
             Connection connection;

             for ( int i = 0; i < evlist.length; i++ ){
                if ( ! (evlist[i] instanceof CallCtlConnAlertingEv) )
                   continue;

                connection = ((CallCtlConnAlertingEv) evlist[i]).getConnection();
                if ( ! connection.getAddress().getName().equalsIgnoreCase("662") )
                   continue;

                try {
                    term.pickup(connection, term.getAddresses()[0]);
                } catch ( Exception e ){
                   System.out.println(e.toString());
                }
             }

}
```

2.1.2.3 Nivel de implementación del estándar por parte de DataVoice.

Los siguientes métodos de están soportados:

- De la clase javax.telephony.callcontrol.CallControlAddress:
  - *boolean CallControlAddress.getDoNotDisturb()*
  - *void CallControlAddress.setDoNotDisturb(boolean)*
  - *boolean CallControlAddress.getMessageWaiting()*
  - *void CallControlAddress.setMessageWaiting(boolean)*
  - *CallControlForwarding[]CallControlAddress.getForwarding()*
  - *void CallControlAddress.setForwarding(CallControlForwarding[])*
  - *void CallControlAddress.cancelForwarding()*

- De la clase javax.telephony.callcontrol.CallControlCall:
  - *Connection CallControlCall.offHook(Address, Terminal)*
  - *void CallControlCall.setConferenceEnable(boolean)*
  - *void CallControlCall.setTransferEnable(boolean)*
  - *Connection CallControlCall.consult(TerminalConnection)*

- De la clase javax.telephony.callcontrol.CallControlConnection:
  - *void CallControlConnection.accept()*
  - *void CallControlConnection.reject()*
  - *void CallControlConnection.addToAddress(java.lang.String)*
  - *Connection CallControlConnection.park(java.lang.String)*

- De la clase javax.telephony.callcontrol.CallControlTerminal:
  - *boolean CallControlTerminal.getDoNotDisturb()*
  - *void CallControlTerminal.setDoNotDisturb(boolean)*
  - *TerminalConnection CallControlTerminal.pickupFromGroup(java.lang.String, Address)*
  - *TerminalConnection CallControlTerminal.pickupFromGroup(Address)*

- De la clase javax.telephony.callcontrol.CallControlTerminalConnection:
  - *void CallControlTerminalConnection.join()*
  - *void CallControlTerminalConnection.leave()*

Los siguientes estados no están soportados:

- De la clase javax.telephony.callcontrol.CallControlConnection:
  - *OFFERED*
  - *DIALING*
  - *OFFERING*

- De la clase javax.telephony.callcontrol.CallControlTerminalConnection:
  - *INUSE*
  - *BRIDGED*

2.1.2.4 Diferencias y comentarios de la implementación de DataVoice con respecto al estándar.

- Sobre un puesto se pueden tener simultáneamente 4 llamadas, una interna, dos salientes y una entrante.

- En llamadas entrantes y salientes, los Connection de la llamada que representan al origen o distinto remoto respectivamente. Son representados por el enlace por el que la llamada entra. Por ejemplo si hay una llamada entrante desde el número de teléfono 999999999, que entra al entorno telefónico de DataVoice por el enlace 169, y va al puesto 662. Entonces la llamada tendrá dos Connection el 169, y el 662 y para obtener el número de teléfono llamante, se utilizará el método Call.getCallingAddress()

- El comportamiento de JTAPI y de sus aplicaciones CTI, se seriamente afectadas en caso de llamadas salientes y entrantes por el tipo de líneas que se utilizan (Digitales o Analógicas). Y es que en líneas analógicas la red da menos eventos que informan a las aplicaciones. Por ejemplo en una llamada saliente si deseamos realizar alguna tarea cuando suene el timbre del destinatario, es muy probable que en muchos casos y dependiendo de la calidad de las líneas analógicas de la red, no se detecte esta situación, influyendo negativamente en las aplicaciones.
  Este problema no depende de JTAPI, y sucede con todos los API de programación. Y el correcto funcionamiento de las aplicaciones dependerá mucho de que el programador conozca este comportamiento para no asumir nada.

- Sobre las transferencias:
  - No se puede realizar ningún tipo de transferencia de una llamada interna.

- Sobre las conferencias:

  - Debido a que cada fabricante de centros de llamadas realiza las conferencias de formas muy distintas los unos de los otros. JTAPI al tener que estandarizarlas, e imponer un comportamiento común. Es complicado que se soporten completamente como impone. A continuación detallaremos el comportamiento del estándar que DataVoice cubre.
    Los terminales de DataVoice poseen un modo de funcionamiento llamado "Modo Conferencia"
    Que supone que cuando un terminal se encuentra en este modo, todas las llamadas que existan en el terminal formarán parte de la misma conferencia. Es decir el audio de todos los dispositivos remotos o locales de las diferentes llamadas se comparte y todos se escuchan.
    Además un terminal al salir del "Modo Conferencia" todas las llamadas dejan de estar en conferencia y dejan de escucharse los unos a los otros.
    Este comportamiento, es diferente del especificado por JTAPI, que no supone que todas las llamadas de un terminal tengan que estar conferenciadas, y tampoco define nada del "Modo conferencia" que es específico de DataVoice.

2.1.3 Paquete javax.telephony.CallCenter

2.1.3.1 Introducción

Con este paquete podremos:

a) Obtener los grupos de agentes.
b) Añadir y eliminar agentes a los grupos de agentes.
c) Obtener información sobre las llamadas que están en un grupo de agentes.
d) Obtener el estado de los agentes.
e) Asociar y recuperar datos a las llamadas.
f) Obtener información sobre los enlaces que utiliza una llamada.
g) Realizar llamadas predictivas.

2.1.3.2 Ejemplos

2.1.3.2.1 Número de agentes de todos los grupos de agentes

```
import javax.telephony.*;
import javax.telephony.events.*;
import javax.telephony.callcenter.*;
import javax.telephony.callcenter.evets.*;

public class NumeroDeAgentesGrupo {

        ////////////////////////////////////////////////////////////////////////

        public static void main(String[] args)      {
                CallCenterProvider       prov;
                JtapiPeer               peer;
                ACDAddress[]            acdAddresses;
                ACDAddressObserver  observer;

                try {

                   // Obtenemos una instancia de la implementación de
                   // JtapiPeer de DataVoice.
                   peer  = JtapiPeerFactory.getJtapiPeer(null);


                   // Obtenemos un proveedor que de el servicio CallCenter.
                   prov  = (CallCenterProvider)
                   peer.getProvider("SERVICE=CallCenter;SERVER=localhost");

                   // Instanciamos el observer que se añadirá a todos los grupos
                   // de agentes.
                   observer = new MiACDAddressObserver();

                   // Obtenemos todos los grupos de agentes definidos.
                   acdAddresses = prov.getACDAddresses();

                   System.out.println("CONFIGURANDO OBSERVERS. ESPERE ....");

                   // Enlazamos el observer a todos los grupos.
                   for ( int i = 1; i < acdAddresses.length; i++ )
                       acdAddresses[i].addObserver(observer);
                   System.out.println("YA ESTA. ");

                   // Observamos por la salida estandar los mensajes que se
                   // producen del observer
                   Thread.sleep(2 * 60 * 1000);


                   // Finalizamos el proveedor
                   prov.shutdown();

                } catch ( Throwable e ){
                     System.out.println(e.toString());
                }

                // Terminamos la aplicación
                System.exit(0);
        }

        ////////////////////////////////////////////////////////////////////////
}
```

```
////////////////////////////////////////////////////////////////////////////
/// Esta clase implementa la interfaz javax.telephony.callcenter.ACDAddressObserver.
/// Imprime por la salida estándar los cambios en el número de agentes que existen en
/// cada grupo.

class MiACDAddressObserver implements ACDAddressObserver {

    ////////////////////////////////////////////////////////////////////////
    //// Implementación del método definido en la interfaz ACDAddressObserver

    public void addressChangedEvent(AddrEv[] eventos){
       ACDAddress    acdAddress = null;
       Agent[]       agents     = null;
       AgentTerminal a;

       // Solo imprimimos en número de agentes que hay en cada grupo cuando el
       // agente se de baja o de alta en un grupo.

       for ( int i = 0; i < eventos.length; i++ ){
              if ( ! ((eventos[i] instanceof ACDAddrLoggedOnEv) ||
                      (eventos[i] instanceof ACDAddrLoggedOffEv)))
                 continue;

              try {
                   // Obtenemos el grupo del agente.
                   acdAddress = (ACDAddrEv)eventos[i]).getAgent().getACDAddress();

                   // Imprimimos el número de agentes que hay en el grupo.
                   if ( (agents = acdAddress.getLoggedOnAgents()) == null){
                        System.out.println("El grupo " + acdAddress.getName() +
                                           " no tiene agentes.");
                        return;
                   }

                   System.out.print("El grupo " + acdAddress.getName() + " tiene " +
                                    agents.length + " agentes. ( ");

                   for ( int j = 0; j < agents.length-1; j++ )
                      System.out.print(agents[j].getAgentID() + ", " );
                         System.out.println(agents[agents.length-1].getAgentID() + ")");
              } catch (javax.telephony.MethodNotSupportedException e ){
                      // Nunca se dará, pues DataVoice lo soporta.
              }
       }
    }

       ////////////////////////////////////////////////////////////////////////
}
```

2.1.3.2.2 Dar de alta a un agente en un grupo.

```
////////////////////////////////////////////////////////////////////////////
/// Da de alta al agente 630 en el grupo 200 trabajando en el puesto 662.

import javax.telephony.*;
import javax.telephony.callcenter.*;

public class LogOnApp {

        public static void main(String[] args){
                JtapiPeer         peer     =  null;
                CallCenterProvider prov     =  null;
                AgentTerminal     term     =  null;
                ACDAddress[]      grupos =  null;
                Agent             agent    = null;

                try {
                   // Obtenemos una instancia de la implementación
                   // de JtapiPeer de DataVoice.

                   peer = JtapiPeerFactory.getJtapiPeer(null);

                   // Obtenemos un proveedor que de el servicio CallCenter.
                   prov = (CallCenterProvider)
                         peer.getProvider("SERVER=PcCarlos;SERVICE=CallCenter");


                   // Obtenemos los grupos
                   grupos = prov.getACDAddresses();

                   // Obtenemos el terminal sobre el que el agente se dará de alta.
                   term = (AgentTerminal) prov.getTerminal("662");

                   // Hacemos que el agente pase a formar parte del grupo 200.
                   agent = term.addAgent(term.getAddresses()[0], grupos[200],
                                         Agent.READY, "630", null);

                   // Finalizamos el proveedor.
                   prov.shutdown();
                } catch ( Exception e ){
                   System.out.println( e.toString() );
                }

                // Terminamos la aplicación
                System.exit(0);
        }
}
```

2.1.3.2.3 Asociar datos a una llamada

```
////////////////////////////////////////////////////////////////////////////////
//// Realiza una llamada desde el 663 al 662 y muestra durante unos segundos los
//// eventos producidos al cambiarse la información asociada a una llamada.
//// Esta información que se asocia son instancias de la clase Cliente.

import javax.telephony.*;
import javax.telephony.callcenter.*;
import javax.telephony.callcenter.events.CallCentCallAppDataEv;

public class AssociateDataCallApp  {

       public static void main(String[] args) {
              JtapiPeer             peer;
              CallCenterProvider    prov;
              CallCenterCall        call;
              CallCenterCallObserver callObserver;
              Connection[]          connections;

              try {
                 // Obtenemos una instancia de la implementación de JtapiPeer de
                 // DataVoice.
                 peer  = JtapiPeerFactory.getJtapiPeer(null);

                 // Obtenemos un proveedor que de el servicio CORE.
                 prov  = (CallCenterProvider) peer.getProvider(
                                              "SERVICE=CALLCENTER;SERVER=localhost);

                 // Solicitamos al proveedor una nueva llamada.
                 call = (CallCenterCall) prov.createCall();

                 // Instanciamos el CallObserver que se enlazará a la llamada.
                 callObserver  = new MyCallCenterCallObserver();

                 // Enlazamos el CallObserver a la llamada.
                 call.addObserver(callObserver);

                 // Realizamos una llamada
                 connections = call.connect(prov.getTerminal("663"),
                                            prov.getAddress("663"), "662");

                 // Asociamos distintos datos a la llamada
                 call.setApplicationData(new Cliente("1111111"));
                 call.setApplicationData(new Cliente("2222222"));
                 call.setApplicationData(new Cliente("3333333"));

                 // Esperamos un tiempo para que de tiempo a imprimirse los cambios
                 // de información de la llamada
                 Thread.sleep(500);

                 // Desenlazamos el CallObserver de la llamada
                 call.removeObserver(callObserver);


                 // Finalizamos el proveedor
                 prov.shutdown();
              } catch ( Exception e ){
                     System.out.println(e.toString());
              }

              // Terminamos la aplicación.
              System.exit(0);
       }
}
```

```
////////////////////////////////////////////////////////////////////////////
/// CallObserver que imprime por la salida estándar el cambio de los datos
/// asociados a la llamada

class MyCallCenterCallObserver implements CallCenterCallObserver {

    ////////////////////////////////////////////////////////////////////
    //// Implementamos el método de la clase CallObserver.

    public void callChangedEvent(javax.telephony.events.CallEv[] events){
       Cliente data;

       // Recorremos el array de eventos e imprimimos los eventos
       // CallCentCallAppDataEv

       for ( int i = 0; i < events.length; i++ ) {
            if ( ! (events[i] instanceof CallCentCallAppDataEv) )
                 continue;

            // Obtenemos los datos asociados a la llamada, que deben
            // ser una instancia de la clase Cliente.

            data = (Cliente) ((CallCentCallAppDataEv)events[i]).getApplicationData();
            System.out.println( data.getDni() );
       }
    }
}



////////////////////////////////////////////////////////////////////////////
/// Las instancias de esta clase son las que se enlazan en este ejemplo como datos a la
/// llamada.

class Cliente implements java.io.Serializable {

       private java.lang.String dni;

       ////////////////////////////////////////////////////////////////////
       /// Constructor

       public Cliente(java.lang.String dni){
              this.dni = dni;
       }

       ////////////////////////////////////////////////////////////////////
       ///  Devuelve el dni del cliente

       public java.lang.String getDni(){
              return this.dni;
       }
}
```

2.1.3.3 Nivel de implementación del estándar por parte de DataVoice.

DataVoice no implementa la funcionalidad de las siguientes clases:

- *javax.telephony.CallCenter.ACDManagerConnection.*
- *javax.telephony.CallCenter.RouteAddress.*
- *javax.telephony.CallCenter.RouteCallback.*
- *javax.telephony.CallCenter.RouteSession.*
- *javax.telephony.CallCenter.ACDManagerAddress.*

Los siguientes métodos no están soportados.

- De la clase javax.telephony.callcenter.CallCenterProvider:
  - *ACDManagerAddress[] CallCenterProvider.getACDManagerAddresses()*
  - *RouteAddress[] CallCenterProvider.getRouteableAddresses()*

- De la clase javax.telephony.callcenter.ACDAddress:
  - *int ACDAddress.getQueueWaitTime()*
  - *int ACDAddress.getRelativeQueueLoad()*
  - *ACDManagerAddress ACDAddress.getACDManagerAddress()*

- De la clase javax.telephony.callcenter.ACDConnection:
  - *ACDManagerConnection ACDConnection.getACDManagerConnection()*

- De la clase javax.telephony.callcenter.AgentTerminal:
  - *void AgentTerminal.setAgents(Agent[])*

2.1.3.4 Diferencias y comentarios de la implementación de DataVoice con respecto al estándar.

- El objeto que se va ha asociar a una llamada mediante el método CallCenterCall.setApplicationData(java.lang.Object) deben implementar la interfaz Serializable del paquete java.io.

- Los datos que son asociados a las llamadas se asocian por valor y no por referencia, es decir, por ejemplo si dos aplicaciones tienen referencia a la misma llamada, y la primera asocia datos a la llamada y la segunda los lee, y la primera asocia de nuevo datos a la llamada, los datos que tiene la segunda aplicación no reflejan este cambio a no ser que los vuelva a pedir.

- Estos datos sólo serán visibles en el entorno JTAPI. Por lo que no podrán ser recuperados o modificados usando otros APIs de DataVoice.

- Hay definidos 256 grupos de agentes (ACDAddress) cuyos identificadores van desde 0 a 255. Además estos están definidos desde que la aplicación comienza. Es decir no se crean ni se eliminan de forma dinámica.

- Un agente puede trabajar a lo sumo 4 grupos de agentes.

- Sólo es posible obtener los grupos a los que pertenecen los agentes, cuando una vez arrancado el servidor, estos se dan de alta en algún grupo. Es decir por ejemplo, si el servidor se arranca hoy a las 15:00 y los agentes estaba trabajando desde las 8:00 entonces sólo tendremos referencia de los agentes que se den de alta en algún grupo a partir de las 15:00.

- Cuando ponga oyentes a un ACDAddress (Un grupo de agentes), observará que si un agente está de alta en n grupos y se da de alta en otro nuevo. Entonces recibirá eventos ACDAddrNotReadyEv, ACDAddrLoggedOffEv de los grupos n grupos en los que estaba antes de recibir los eventos ACDAddrLoggedOnEv y ACDAddrReadyEv de los n grupos en los que estaba y del nuevo.

- Cuando ponga oyentes a un AgentTerminal , observará que si un agente está de alta en n grupos y se da de alta en otro nuevo. Entonces recibirá eventos AgentTermNotReadyEv,

AgentTermLoggedOffEv de los grupos n grupos en los que estaba antes de recibir los eventos AgentTermLoggedOnEv y AgentTermReadyEv de los n grupos en los que estaba y del nuevo.

- En llamadas predictivas, el connection origen se crea solamente cuando el destino descuelga. Esto puede suponer un problema sobre líneas analógicas, pues no siempre se detecta el descolgado.

- En llamadas predictivas, el número de timbres que se deben producir antes de que la llamada se asuma cómo no contestada no es. Es decir, si indico que una llamada predictiva con 6 timbres no ha sido contestada, en realidad, se podrán dar un número de timbres aproximado, 5 o 7.

- Los enlaces (CallCenterTrunk) son utilizados tanto para llamadas salientes como para entrantes. Por lo que el método int CallCenterTrunk.getType() para un mismo enlace, podrá devolver valores diferentes dependiendo si la llamada a la que está asociada es entrante o saliente.

2.1.4 Paquete javax.telephony.Phone

2.1.4.1 Introducción

Con este paquete podremos:

a) Obtener el display de un puesto.
b) Obtener y configurar el volumen de un puesto.
c) Obtener el estado de los Leds de un puesto.
d) Simular la pulsación de los botones de un puesto.

2.1.4.2 Ejemplos

2.1.4.2.1 Observar el display de un puesto

```
////////////////////////////////////////////////////////////////////////////
/// Este ejemplo muestra durante unos segundos el display del puesto cuyo nombre
/// se pasa como argumento

import javax.telephony.*;
import javax.telephony.phone.*;
import javax.telephony.events.TermEv;
import javax.telephony.phone.events.DisplayUpdateEv;

public class PhoneTerminalDisplayApp {

        public static void main(String[] args){
                JtapiPeer       peer     = null;
                Provider                prov     = null;
                Terminal                term     = null;
                TerminalObserver observer = null;

                if ( args.length != 1 ){
                     System.out.println("Formato: PhoneTerminalDisplayApp <puesto>");
                     System.exit(1);
                }

                try {
                        peer   = JtapiPeerFactory.getJtapiPeer(null);
                        prov   = peer.getProvider(null);

                        term     = prov.getTerminal(args[0]);
                        observer = new MyPhoneTerminalObserver();

                        term.addObserver(observer);
                        Thread.sleep(30 * 1000);
                        term.removeObserver(observer);

                        prov.shutdown();
                } catch ( Exception e ){
                        System.out.println(e.toString());
                }

                System.exit(0);
        }
}

class MyPhoneTerminalObserver implements TerminalObserver {
        public void terminalChangedEvent(TermEv[] ev){
                for ( int i = 0; i < ev.length; i++ ){
                        if (! ( ev[i] instanceof DisplayUpdateEv) )
                                continue;
                        System.out.println(((DisplayUpdateEv) ev[i]).getDisplay(0, 0));
                }
        }
}
```

2.1.4.2.2 Subir el volumen del altavoz de un puesto.

```
////////////////////////////////////////////////////////////////////////////
/// Este ejemplo sube el volumen del Speaker del terminal local 662 al máximo.

import javax.telephony.*;
import javax.telephony.phone.*;

public class UpVolumeFullApp {

   public static void main(String[] args){
         JtapiPeer        peer  = null;
         Provider         prov  = null;
         PhoneTerminal    term  = null;
         ComponentGroup[] groups= null;
         Component[]      c     = null;

         try {
              // Obtenemos un proveedor que de el servicio Phone
              peer = JtapiPeerFactory.getJtapiPeer(null);
              prov = peer.getProvider("SERVICE=Phone;SERVER=localhost");

              // Obtenemos el terminal local 662
              term = (PhoneTerminal) prov.getTerminal("662");

              groups = term.getComponentGroups();
              for ( int i = 0; i < groups.length; i++ ){
                  if (! (groups[i].getDescription().equalsIgnoreCase("PhoneSpeaker")) )
                       continue;

                  // Estamos en el grupo del Speaker
                   c = groups[i].getComponents()[0];
                  ((PhoneSpeaker) c.setVolume(PhoneSpeaker.FULL);
                   break;
                }

               prov.shutdown();
         } catch ( Exception e ){
            System.out.println(e.toString());
         }

         System.exit(0);
   }

}
```

2.1.4.2.3 Presionar las teclas de un puesto.

```
////////////////////////////////////////////////////////////////////////////
/// Este ejemplo muestra como realizar una llamada interna del 663 al 662 usando
/// PhoneTerminal deja que suene durante un tiempo y luego finaliza la llamada.

import javax.telephony.*;
import javax.telephony.phone.*;

public class PhoneTerminalPressKeyApp {

   private static void PressKey(Component[] botones, String buttonID){
      PhoneButton btn;

      for ( int i = 0; i < botones.length; i++){
         btn = (PhoneButton) botones[i];
         if ( ! btn.getName().equalsIgnoreCase(buttonID) )
               continue;
         btn.buttonPress();
         return;
      }
   }

   public static void main(String[] args){
      JtapiPeer          peer    = null;
      Provider           prov    = null;
      Terminal           term    = null;
      ComponentGroup[] groups    = null;
      Component[]        botones = null;

      try {

         peer   = JtapiPeerFactory.getJtapiPeer(null);
         prov   = peer.getProvider("SERVICE=Phone");
         term   = prov.getTerminal("663");

         groups = ((PhoneTerminal) term).getComponentGroups();
         for ( int i = 0; i < groups.length; i++ ){
               if ( "PhoneButton".equalsIgnoreCase(groups[i].getDescription())){
                      botones = groups[i].getComponents();
                      break;
               }
         }

         PhoneTerminalPressKeyApp.PressKey(botones, "<=>");  // Llamada Interna
         PhoneTerminalPressKeyApp.PressKey(botones, "6");
         PhoneTerminalPressKeyApp.PressKey(botones, "6");
         PhoneTerminalPressKeyApp.PressKey(botones, "2");

         Thread.sleep(1000);

         PhoneTerminalPressKeyApp.PressKey(botones, "Disconnect");
         prov.shutdown();
         System.out.println("Fin del ejemplo");
      } catch ( Exception e ){
         System.out.println(e.toString());
      }

      System.exit(0);
   }
}
```

2.1.4.3 Nivel de implementación del estándar por parte de DataVoice.

DataVoice no implementa la funcionalidad de las siguientes clases:

- javax.telephony.phone.*PhoneGraphicDisplay*
- javax.telephony.phone.*PhoneHookswitch*
- javax.telephony.phone.*PhoneRinger*

2.1.4.4 Diferencias y comentarios de la implementación de DataVoice con respecto al estándar.

La especificación estándar de JTAPI para este paquete deja mucho que desear. Por lo que a continuación se detallará como está implementado este paquete por nuestra implementación.

Los puestos devuelven en el método *ComponentGroup[] PhoneTerminal getComponentGroups()* un array de ComponentGroup que se detalla a continuación:

COMPONENTGROUP.GETDESCRIPTION()	DESCRIPCIÓN DE LO QUE REPRESENTA EL GRUPO
PhoneButton	Representa el conjunto de botones del puesto.
PhoneDisplay	Representa el display del puesto.
PhoneSpeaker	Representa el micrófono del puesto.
PhoneLamp	Representa el conjunto de Leds del puesto

Los botones de los puestos Dharma tienen unos identificadores únicos que debemos conocer para poder trabajar con ellos.

IDENTIFICADOR	DESCRIPCIÓN
*	Botón de rellamada.
C	Botón de conferencia.
D	Botón que permite la grabación de llamadas.
0	Botón cuya etiqueta es un 0.
1	Botón cuya etiqueta es un 1.
2	Botón cuya etiqueta es un 2.
3	Botón cuya etiqueta es un 3.
4	Botón cuya etiqueta es un 4.
5	Botón cuya etiqueta es un 5.
6	Botón cuya etiqueta es un 6.
7	Botón cuya etiqueta es un 7.
8	Botón cuya etiqueta es un 8.
9	Botón cuya etiqueta es un 9.
Disconnect	Botón que permite finalizar la comunicación.

F	Botón para la activación de funciones.
F1	Tecla de función F1.
F2	Tecla de función F2.
F3	Tecla de función F3.
F4	Tecla de función F4.
F5	Tecla de función F5.
F6	Tecla de función F6.
#	Tecla de funciones especiales. Y cuya etiqueta es #.
Inbound	Tecla de la llamada entrante.
<=>	Tecla de llamada interna.
Micro	Tecla de manos libres.
Mute	Silenciador
Outbound A	Tecla de llamada saliente A.
Outbound B	Tecla de llamada saliente B.
Alt	Tecla de altavoz.
T	Tecla de transferencia.
Volume Down	Tecla para bajar el volumen del puesto.
Volume Up	Tecla para subir el volumen del puesto.

Los Leds de los puestos Dharma tienen unos identificadores únicos que debemos conocer para poder trabajar con ellos.

IDENTIFICADOR	DESCRIPCIÓN
Alt	Led de Altavoz y Mute.
C	Led de conferencia.
Micro	Led del manos libre.
<=>	Led de llamada interna.
Outbound A	Led de llamada saliente A.
Outbound B	Led de llamada saliente B.
F	Led de función
Lib	Led de operador libre.
Act	Led de operador activado
Esp	Led de llamada en espera.
Disconnect	Led de finalización de la comunicación.
Inbound	Led de llamada entrante.

2.2 Instalación de los paquetes para desarrollar aplicaciones basadas en JTAPI.

El fichero *jtapi.jar* (Java Archive) que contiene la especificación JTAPI podrá obtenerse en la siguiente dirección http://java.sun.com/products/jtapi/index.html.

La implementación de JTAPI de DataVoice se encuentra en el fichero *DVJtapiClient.jar* que es suministrado.

Una vez instalado los ficheros en un disco duro local o remoto, deberá configurar la variable de entorno CLASSPATH para que apunte a los ficheros *jtapi.jar* y *DVJtapiClient.jar*.

Por ejemplo, en caso de una máquina Windows 9x y suponiendo que el paquete se encuentra en el directorio c:\windows\java\packages de la máquina. Esta variable de entorno puede ser configurada en el fichero c:\autoexec.bat de la siguiente manera:

SET CLASSPATH=%CLASSPATH&;c:\windows\java\packages\DVJtapiCliente.jar.

2.3 Fichero de configuración de las aplicaciones clientes.

Como podrá observar en las aplicaciones de ejemplo anteriores. En la instrucción cuya finalidad es pedir el proveedor adecuado. *Provider JtapiPeer.getProvider(String)* en muchos casos este parámetro es null. Pues bien en estos casos, o en los casos en el que el usuario no de explícitamente valores a todos los parámetros, estos son leidos de un fichero de configuración que debe existir en el directorio que indique la propiedad *java.home* y cuyo nombre es *DVJtapi.conf.*

Nota: Podrá obtener este directorio mediante la siguiente aplicación:

```
public class ObtenerJavaHomeDir {
        public static void main(String args[]){
                System.out.println(System.getProperty("java.home") );
        }
}
```

El formato del fichero es que en cada línea de este debe haber una propiedad, por ejemplo:
SERVER=pcServidorJTAPI
SERVICE=CallControl

La propiedad SERVER indica a las aplicaciones clientes que no especifiquen explícitamente un valor para esta propiedad, que el servidor que les atenderá está ejecutándose en la máquina cuyo nombre o dirección IP está detrás del símbolo igual. (En el ejemplo, pcServidorJTAPI)

La propiedad SERVICE indica a las aplicaciones clientes que no especifiquen explícitamente un valor para esta propiedad, que el servicio que solicitan es el que se encuentra después del símbolo igual. (En el ejemplo CallControl).

3. Instalación del servidor.

3.1 Introducción

El servidor JTAPI de DataVoice, dará servicio a los clientes que lo soliciten. Será el que esté en contacto directamente con la plataforma de telefonía usando para ello una serie de componentes privados de DataVoice.

3.2 Requerimientos necesarios para instalar el servidor

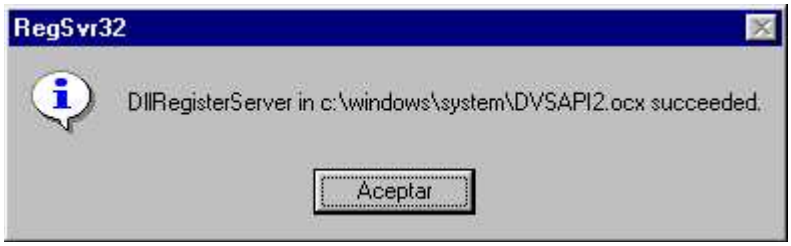
Debe ser instalado en máquinas que ejecuten un sistema operativo Windows de 32 bits, Windows 9x, 2000, NT, XP, etc.

Deben estar conectado en Red y ser visibles por las máquinas en las que se vayan a ejecutar software CTI basado en Jtapi.

Instalar los paquetes *DVJTAPIServer.jar* y *jtapi.jar* en un directorio que sea visible mediante la variable de entorno CLASSPATH.

Instalar y registrar el componente DVSAPI2.ocx versión 1.0.1 o superior.

Para registrar este componente se deberá usar la aplicación regsvr32.exe que viene de serie con el sistema operativo. Y la forma de registrarlo es: "*regsvr32 <rutaAlFicheroOCX>*"
Por ejemplo: *regsvr32 c:\windows\system\dvsapi2.ocx*. Debiendo aparecer a continuación un cuadro de dialogo como el siguiente:



3.3 ¿ Cómo ejecuto el servidor ?

Para poner en marcha el servidor, debemos ejecutar el siguiente comando:
jview DataVoice.jtapi.Servidor.StartJtapiServer [<CTIServerAddress>< CTIServerPort >]

CTIServerAddress: Nombre o dirección IP de la máquina en la que corre el servidor CTI.
CTIServerPort: Puerto en el que está escuchando el servidor CTI.

Una vez arrancado el servidor deberá aparecer una mensaje indicando que el servidor está correctamente arrancado.

3.4 ¿ Cómo paro el servidor ?

Una vez arrancado el servidor, pulsando la tecla 'S'. Este se para y las aplicaciones clientes realizarán la tarea que tengan programada para esta situación.

3.5 ¿ Puedo arrancar varios servidores ?

Se puede arrancar un servidor por máquina, por lo que si tenemos 5 máquinas con sistemas operativos Win32 podremos arrancar 5 servidores.

4. Consejos de programación con JTAPI.

1. Es importante que los observers y listeners añadidos a los objetos Address, ACDAddres, Call, Provider, Connection, TerminalConnection, Terminal, etc. se eliminen antes de que la aplicación termine. Pues mantendrán al servidor limpio y podrá ofrecer mejor servicio a las otras aplicaciones

2. Evitar los bucles cerrados, pues restarán valiosos recursos a las demás aplicaciones por hacer menos eficiente al servidor atendiendo a las continuas peticiones de dentro del bucle.
   Por ejemplo. Imagine que desea realizar una aplicación que finalice una llamada cuando el Connection destino de esta tenga el estado Connection.Alerting.

   Pues bien en vez de estar en un bucle similar al siguiente:

```
.... .... ....
Call call = null;
Connection[] connections = null;
.... .... ....
connections = call.connect(.......);
while ( connections[1].getState() != Connection.Alerting )
        ;
connections[1].disconnect();
.........
```

   Es conveniente que añada un observer o listener al Connection destino y cuando se reciba el evento correspondiente finalice la llamada.

3. Poner siempre un listener u observer al Proveedor para que en caso de que esté deje de estar en estado de servicio, realizar un tratamiento adecuado.